TU WIEN Informatics

Interoperabilitätsanalyse der Metamodellierungsframeworks ADOxx und EMF

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Konstantinos Anagnostou, B.Sc.
Matrikelnummer 11736297

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ass. Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork
Mitwirkung: Univ.-Prof. Mag. Dr. Manuel Wimmer

Wien, 21. April 2021

Konstantinos Anagnostou | Dominik Bork

Informatics

Interoperability Analysis of the Metamodel Frameworks ADOxx and EMF

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Konstantinos Anagnostou, B.Sc.
Registration Number 11736297

to the Faculty of Informatics

at the TU Wien

Advisor: Ass. Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork
Assistance: Univ.-Prof. Mag. Dr. Manuel Wimmer

Vienna, 21st April, 2021

Konstantinos Anagnostou   Dominik Bork

Erklärung zur Verfassung der Arbeit

Konstantinos Anagnostou, B.Sc.

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

Wien, 21. April 2021

Konstantinos Anagnostou

Danksagung

Zuallererst möchte ich mich bei meinem Betreuer Ass. Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork bedanken, welcher mich von der Themenfindung, über die Konzeption und Implementierung, bis hin zur Abgabe der Diplomarbeit stets unterstützt hat. Vielen Dank auch für das Entegegenkommen und die Flexibilität in diversen Aspekten. Zudem möchte ich mich für die Chance bedanken, das akademische Leben abseits der Diplomarbeit erleben zu dürfen, speziell für die Teilnahme an der Konferenz, als auch für das Angebot als wissenschaftlicher Mitarbeiter weiterhin am Institut mitzuwirken.

Des weiteren möchte ich meinem zweiten Betreuer Univ.-Prof. Mag. Dr. Manuel Wimmer danken, welcher trotz der Entfernung und der Beschränkung von Treffen auf die virtuelle Welt, stets viel Untestützung und schnelle Hilfe bei diversen Problemen geboten hat.

Meiner Familie, speziell meinen Eltern als auch meinen Brüdern Ewa und Vasili möchte ich an dieser Stelle für die seelische und moralische Unterstützung danken, welche mir trotz der großen Distanz immer geboten wurde.

Zu guter Letzt möchte ich mich bei meinen Freunden Markus, Korni, Niki, Verena, Reza, Alex und Triinu, sowie meinen Arbeitskollegen Jakob und Sherif bedanken, mit welchen ich in den letzten Semestern viel Glück und Freude, als auch das eine oder andere Mal (aber hoffentlich nicht zu oft) meinen Frust teilen konnte. Vielen Dank.

Acknowledgements

First, I would like to thank my supervisor, Ass. Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork, who always helped me in this thesis, from finding a topic to conceptualizing and implementing the solution, over to submitting the final draft. Special thanks for the cooperation and the flexibility in various aspects. I would also like to give thanks for the opportunity to experience the academic life besides this thesis, from participating in the conference to the offer of working at the institute as a scientific assistant.

I would then like to thank my second assistant Univ.-Prof. Mag. Dr. Manuel Wimmer, who, despite the distance and the online-only limitation, gave lots of advice and offered quick help when problems arose.

I would also like to thank my family, especially my parents and my brothers Ewa and Vasili, who offered emotional and moral help despite the great distance.

Finally, I would like to thank my friends Markus, Korni, Niki, Verena, Reza, Alex, and Triinu, as well as my colleagues Jakob and Sherif, whom in the last semesters I could share happiness and joy, but also sometimes (hopefully not too often) my frustration with. Thanks a lot!

Kurzfassung

Model Driven Engineering (MDE) ist eines der am häufigsten verwendeten Programmierparadigmen, welches sich mit dem Erstellen und dem Modifizieren von Modellen befasst, was große Flexibilität in der Entwicklung und der Nutzung von Modellartefakten ermöglicht. Zwei bekannte Metamodellierungsframeworks sind ADOxx und das Eclipse Modeling Framework (EMF). Beide dieser Tools beinhalten ein weit gefächertes Repertoire an Metamodellierungstechniken und mächtige Modellumgebungen. Sie sind in vielerlei Hinsicht ident zueinander, besitzen aber auch einige Unterschiedlichkeiten in fundamentalen Implementierungsansätzen und Nutzungsweisen.

Bisher existieren diese beiden Plattformen in Isolation voneinander. Es ist nicht möglich die Metamodelle beider Plattformen auszutauschen, das bedeutet es gibt keine Möglichkeit ein Metamodell aus einer Plattform in die andere zu importieren und dort weiterzuverwenden. Solch ein Feature würde es unterschiedlichen Domainexperten und Modellentwicklern ermöglichen, gemeinsam an einem (Meta)modell zu arbeiten, ohne sich dabei auf eine gewisse Plattform beschränken zu müssen.

Diese Thesis beantwortet die Frage, ob solch eine Interoperabilität zwischen den beiden Plattformen möglich ist und wie sie implementiert werden kann. Es wird sich zeigen, dass Interoperabilität zwischen den beiden Modellierumgebungen für die meisten Metamodelle tatsächlich möglich ist. Zudem wird sich zeigen, dass der erarbeitete Lösungsansatz sowohl syntaktisch als auch semantisch valide ist, also gültige Metamodelle für die Ziel Plattform erstellt werden können.

Dazu werden zuerst die beiden Metamodellierungsplattformen miteinander verglichen und deren Gemeinsamkeiten und Unterschiede ausgearbeitet. Danach wird ein Konzept erstellt, mit dem man eine Brücke erstellen kann, welche Metamodell Dateien von einer Plattform in die andere überführen kann. Diese Brücke wird anschließend implementiert und im letzten Schritt mit Hilfe einer Evaluation auf Basis von unterschiedlichen Metamodellen auf syntaktische Korrektheit und semantische Äquivalenz validiert.

Abstract

Model-Driven Engineering (MDE) has become a pivotal way to conduct software engineering, focusing on model creation and modification, introducing great flexibility in development and operation. Two major metamodeling platforms used in this context are ADOxx and the Eclipse Modeling Framework (EMF). Both tools offer a great set of metamodeling techniques and environments. They share a lot of similarities but also have differences in crucial aspects of their implementation and usage.

As of now, both these two platforms exist in isolation. It is impossible to use the two platforms interchangeably, i.e., take a metamodel or model from one platform and use it in the other. With this feature enabled, domain experts and model engineers could work on the same (meta)model, independently of their desired choice of tool.

This thesis focuses on answering whether and how a transformation procedure can enable interoperability between the two platforms. It will show that interoperability between the two is feasible for most of the given input metamodels and that the proposed solution results are syntactically and semantically valid.

This is done by first analyzing the differences and similarities of the two platforms and then creating a concept for a bridge that maps metamodel files from one platform to another. Later, the implementation of the bridges for both directions is performed, and the evaluation results are analyzed in the context of syntactic and semantic equivalence based on various metamodels.

Contents

CHAPTER 1

Introduction

This section will introduce to the main topic, goals and structure of this thesis. First it will give a motivation and explain, what the current problem is, how the idea arose to find a solution and what benefits the research will have for model developers and researchers. The following section then introduces the goals of this thesis and mentions the concrete research questions. Finally, an explanation of the structure follows, that describes the contents of each individual chapter.

1.1 Motivation

Metamodeling platforms play a vital role in developing model-based applications that can be used in various ways, from model transformation to code generation. Two prominent solutions of metamodeling platforms are ADOxx and the Eclipse Modeling Framework (EMF). The metamodeling platform ADOxx is best known for its advanced graphical components, which allows to create intuitive and fast models. EMF, on the other hand, is known for its strong bondage with the programming language Java, providing a familiar base for such developers to create (meta)models, as well as providing advanced features like model-to-code transformations.

Those platforms show similarities in many ways, like sharing similar first-class concepts or attribute types. They also differ in many ways, from composition concepts to inheritance patterns. While using one of these platforms only extensively provides a good solution to create model-driven applications, the need for interoperability arises. This feature would create a lot of benefits, like giving developers a way to exchange their (meta)models with experts of the other platforms, increasing overall productivity.

"Interoperability extends the border of already existing systems and enables the connection to other systems."[Ker16]. With the help of enabled interoperability between these two platforms, the dependence of a model or domain engineer to a single platform vanishes: It is possible for teams to collaborate in big and complex model-driven projects, without the need to collectively work with the same environment. Different users may be familiar to different tools; organizations may only offer a certain solution of modeling environments. These and many other factors contribute to the usage of various metamodel environments. If two platforms would be interoperable, the teams and individuals could continue working with their desired environment throughout the development and maintaining process, on a single metamodel or model project.

Another benefit of interoperability is the fact, that the best tool can be used for the job. Metamodeling platforms differ in their implementation and some of their key concepts, like inheritance or composition patterns. A sub part of a metamodel could benefit from the features of one platform, while another task could benefit from features of the other. Also, the creation of metamodels and models can differ in either platform. Some may offer advanced features for users to create metamodel elements with a graphical UI. Other platforms may offer an advanced API that allows to build metamodels and models programmatically. With interoperability enabled, an individual semantic subsection can be developed with one platform, while it is then transformed to a model of the target platform, where the remaining part can be implemented. This way, the best features of both metamodeling platforms can be used to create a desired metamodel, which is only possible when having interoperability between them enabled.

As of now, both of the platforms ADOxx and EMF exist in isolation, allowing no exchange of models or metamodels between the two in an intuitive way, other than rebuilding those (meta)models manually in the other platform. Kern and Kühne showed in [KK07], that such model interoperability is indeed possible between ARIS and EMF and that it can create benefits for the users of either platform when they can transform and later use their created models in the other platform.

This thesis focuses on mitigating the problem of missing interoperability between ADOxx and EMF by describing first the problem, proposing a transformation bridge as a solution to interoperability, creating a prototype, and finally analyzing it.

1.2 Goals

This scientific work aims to reveal the possibilities, difficulties, and limitations of interoperability of metamodeling platforms in the concrete context of ADOxx and EMF. The thesis will show differences, similarities, and a procedure to enable interoperability between these two platforms. The findings and conclusions of this task can then be used to derive knowledge in the more general context of interoperability in metamodeling platforms.

This scientific research incorporates creating a programmatic tool that enables interoperability for a model engineer, who will be able to easily transform a (meta)model of one platform to another. Another goal of this work is to create an evaluation based on semantic and syntactic categories, which will further prove the feasibility and performance of this interoperability proposal.

Concretely, the goal of this thesis is to answer the following research questions:

1. **Is interoperability between the two metamodel platforms feasible?**
2. **If interoperability is feasible, how well performs the proposed solution syntactically and semantically?**

1.3 Structure

In chapter 2, we will give an introduction to the critical topics of this thesis. This includes an introduction to metamodeling and model creation, followed by an introduction of the metamodeling platforms ADOxx and EMF, and rounded up with a short dive into interoperability, where we will cover its definition. These foundations should help build a knowledge basis for the rest of the work to follow.

In chapter 3, a focus on different research areas connected to this work will be provided. Similar work will be mentioned, and a representation of the state-of-the-art. Also, the question will be answered whether this type of interoperability analysis has been conducted before.

In chapter 4, a comparative analysis of the metamodeling frameworks ADOxx and EMF will follow, which will build a basis for the metamodel transformations that are to come in the following chapters. This analysis will reveal differences and similarities between the two by categorizing language features and comparing them.

Chapter 5 will then describe the metamodel transformation. The whole process of transforming one metamodel to another will be covered from the used technologies to the concept, over to the actual mapping and implementation details. In section 5.2, the transformation from ADOxx to EMF metamodels will be covered, in section 5.3 the other way will be described. This chapter's last section 5.4 will show, based on an example scenario, the structure and details of the generated artifacts.

In chapter 6, an evaluation of the performed transformations will follow. Thereby, the two main categories for evaluation, namely semantic and syntactic evaluation, will be examined, and the subcategories of the evaluation will be later explained. Finally, the results will be collected and interpreted.

The chapter 7 will focus on the future work that might be conducted with unclear or unfinished topics that arose during the writing of this thesis.

The final chapter of this thesis is chapter 8, which contains a conclusion of the work as well as the results of the evaluation. The research questions that were introduced in the beginning of this thesis are reintroduced and answered.

Additionally, an appendix is provided containing the Dev Ops Manual, which will serve as a code documentation and execution manual for the created code artifacts throughout this thesis.

CHAPTER 2

Foundations

This chapter introduces the software development concept of metamodeling and gives insight into the metamodeling platforms ADOxx and EMF. It helps to understand the primary software development paradigm analyzed in this thesis and the used technologies, which builds a knowledge basis for the rest of this work.

Metamodeling is an integral part of this thesis, building the foundation of an interoperability analysis between metamodeling platforms. Therefore, it describes thoroughly the concept of modeling, the ideas of model abstractions (i.e., metamodeling) and its advantages, the different types of metamodeling and also model transformation, the key research area of this scientific work.

ADOxx and EMF are then described based on their characteristics, strengths, and usages in real-world scenarios. For each metamodeling platform, an analysis follows that sheds light on how metamodels and models are practically created. Then the metamodels of either platform are described, which gives a deep knowledge of the insides of either platform and function as a basis for the comparative analysis in chapter 4.

Finally, the foundations chapter concludes with a definition of interoperability, while also describing certain concerns in context to this scientific work.

2.1 Metamodeling

2.1.1 Modeling

Modeling is the central part of Model-Driven Software Engineering (MDSE), which represents creating models to create programs and environments in the domain of computer science for production use and research. Models can be interpreted as an abstraction of phenomena from the real world [BCWB17]. The phenomena that can be abstracted are theoretically limitless: a model could be an abstraction of the hierarchy of the animal kingdom (abstracting static behavior of real-world phenomena), abstracting processes in a manufacturing line of a car company, or simulating the behavior of weather phenomena like a tornado (both of which represent abstractions of the dynamic behavior of real-world phenomena).

Abstraction, which is an equivalent term when it comes to modeling, can be defined as "[...] the capability of finding the commonality in many different observations and thus generating a mental representation of the reality [...]" [BCWB17]. Finding the commonalities implies that an abstraction always only consists of a part of reality and can never fully represent the complex behavior of the real world.

Even though abstractions cannot fully represent real-world phenomena, they are still suited to understand them better. The limited information about the natural world they contain allows us humans to better process this information and derive knowledge from it, as they are abstracting away from irrelevant aspects, thereby focusing on the relevant ones. Therefore the concept of modeling is well suited to solve problems in computer science alongside classical approaches. Modeling a domain system may help to outline the structure and the behavior of the project, allow non-tech experts to participate in the software engineering process, and with the help of code generation, speed up the overall development process.

2.1.2 Modeling Layers

Following the fact that a model is an abstraction of real-world phenomena, a model can also be abstracted by another model. This abstracted model would describe how a model is defined, what elements and arrangements are allowed to be created, and what operations can be performed upon it. Such a model of a model is called a *metamodel*, since it describes structure and behaviour *about* another model (from the Greek word *meta* meaning *by means of*).

Abstracting a metamodel even further would yield a *meta-meta model*, which would describe the characteristics and constraints of a metamodel. In [BCWB17], Brambilla et al. suggested that going with this abstraction procedure beyond the meta-meta model would be possible (up to infinite instances in theory); in practice, a higher abstraction level than the meta-meta model layer is not necessary since further abstractions can be defined by the meta-meta model itself (for example MOF, which is a meta-meta model can be described by itself using MOF elements).

In general it can be said, that a lower level of modeling always conforms to a higher level of modeling. The Figure 2.1 illustrates these concepts by using an example scenario (a zoo infrastructure), represented by a model and accompanied by its different metamodeling layers.

On the lowest layer, the M0 layer, the model instances of the real world objects and phenomena reside. In this scenario, *Zoo1* is an instance of the model element Zoo, which contains a multitude of animals (*Lion1*, *Lion2* and *Tiger1*), which live in a distinct enclosure (*Enclosure1* and *Enclosure2*).

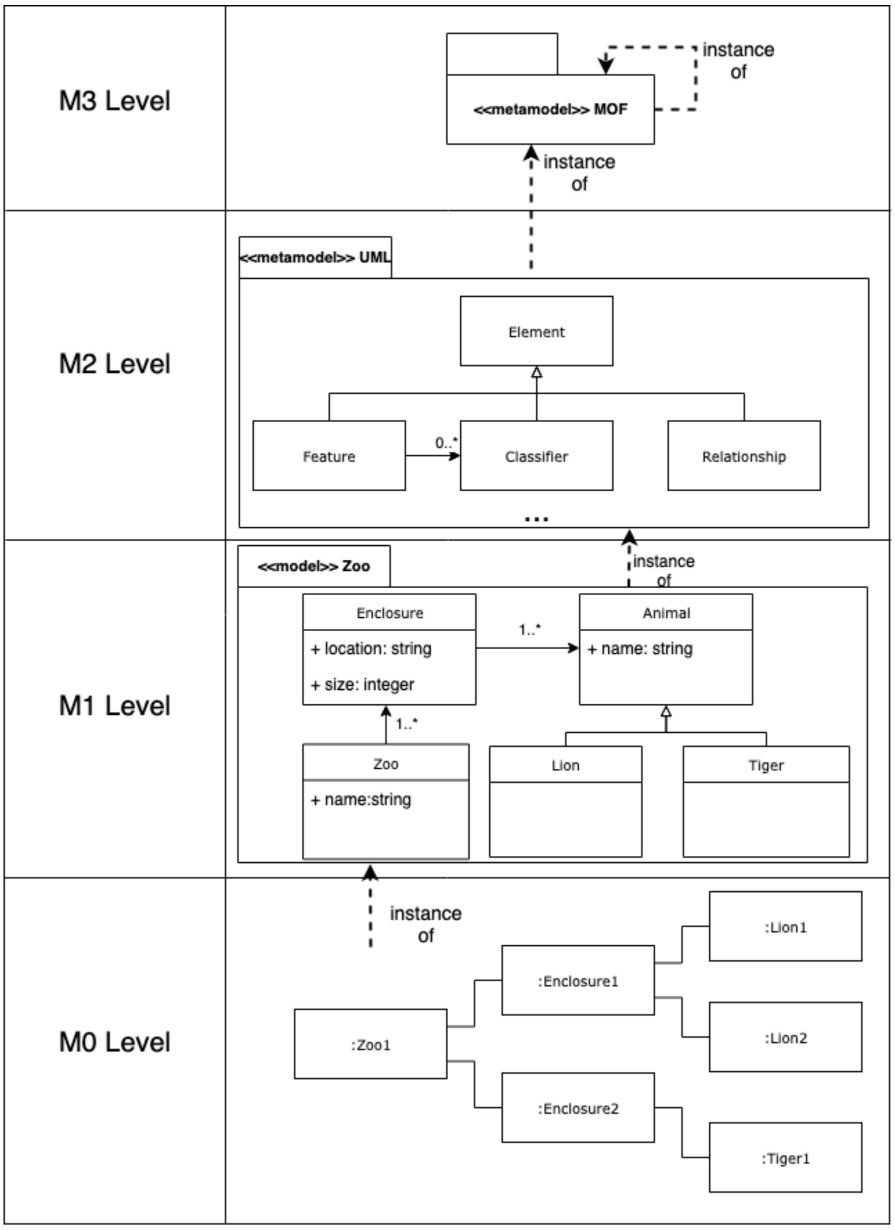


Figure 2.1: Visual representation of metamodel layers of UML

The Model Layer M1 contains the UML definition of the allowed combinations of elements that a Zoo instance can have. Therefore, the model instance on the M1 layer conforms to the metamodel instance on the M2 layer. As can be seen, Zoo, Animal, and Enclosure are represented as UML class elements, while the relationships between them are represented as UML relationship elements, along with different cardinalities.

Going one layer further, the M2 layer, or metamodel layer, defines all the elements of

UML and their relationships. As noted in [cs.], those consist among others of *Feature*, *Classifier* and *Relationship*, which are sufficient to represent the given model.

On the M3 layer, the meta-meta model layer is defined, which describes the different elements and constraints a UML instance can have. It shows that UML conforms to the meta-meta model of the Meta-Object Facility (MOF). The meta-meta model, in this case, uses the elements from MOF to describe itself, therefore, showing that MOF is recursively defined and another layer above the M3 layer is not required.

2.1.3 Creating metamodels

When it comes to metamodeling, there are different ways to create a metamodel and therefore create model instances out of it. Graphical metamodel platforms are the standard in creating metamodels. They come with lots of advantages for the metamodel editor and the model user.

Graphical metamodeling tools include ADOxx and EMF, which will be described in sections 2.2 and 2.3 respectively. Those tools use graphical elements or pre-defined structures to create metamodels. EMF uses a graphical metamodeling interface that allows the user to drag and drop, create and edit certain elements to a canvas, and build the metamodel incrementally by using a UML-like graphical syntax. ADOxx allows for creating metamodel elements within the metamodel editor, consisting of windows and various input fields, where metamodel elements are represented in a graphical, hierarchical way. The authors in [BKP20] conducted various research in graphical metamodeling platforms and categorized their structure, specification and notation. They showed that common structures of modeling languages include slicing, i.e., separating the elements in different semantic or syntactic chunks, referencing, i.e., linking metamodel elements by name to avoid redundancy and matrices, i.e., an environment that supports the creation and editing of table-based metamodels to support the idea of relatedness and structuredness for the metamodel editor.

Key aspects of this work also include the possible serialization formats that the metamodel environments support. To save the state of metamodels or share them with other parties, the user chooses to serialize them into a specific format. The specific metamodel environment usually limits the capabilities of a particular format. The authors in [BKP20] show that the compatibility, readability, and extendability of those formats differ for each platform. Therefore the serialization specification of a metamodel and model artifact plays a vital role in metamodel and model interoperability. Model files of a source platform must be accessible to process their information, and metamodel files must be adjustable to create files with the correct information for the target platform.

2.1.4 Model Transformations

Model-driven development is more advantageous than non-model-driven development because developers can perform model transformations on a particular model. Model transformations procedures can transform a model into another model by describing specific transformation rules that are either defined by the developer (model mapping), derived automatically by creating rules from an external system, or by using the principle of "everything is a model" where the transformation itself can be described as a model that conforms to a shared metamodel of all involved models. [BCWB17]

Model transformations are used to alter certain elements or model behavior in an easy and preferably automated way. This can help the developer or user of a model to implement changes that arose due to changed requirements.

There are differences in how model transformation procedures can perform such a transformation:

On the one hand, transformations can be performed on the same level of context, i.e., concrete elements of the same model are altered and changed to other elements that are part of a shared metamodel. Taking the Zoo scenario as an example: a model transformation could be performed by replacing all lions in the model instance with tigers. On the other hand, a transformation does not have to be bound to the same context. EMF, for example, allows for the creation of a complete and valid Java model out of a given Ecore model that represents the same model but is part of different scope, i.e., no longer within EMF but within the scope of the Java programming language.

Since metamodels are models themselves (as they are just a higher abstraction of a specific model), certain transformation procedures can transform them in the same way. As will be seen in chapter 5, this scientific work heavily focuses on transforming metamodels from one environment (ADOxx or EMF) to the other, therefore representing a transformation that is not bound to the same context of the source metamodel.

2.2 ADOxx

2.2.1 Introduction

ADOxx is mostly used nowadays in the academic context in order to create rich-feature modeling tools [BWA21]. It is part of OMiLAB, "an open digital ecosystem designed to help one conceptualize and operationalize conceptual modeling methods"[BBK$^+$19], which helps to perform scientific and academic research in the context of model engineering, gaining insights in this area and benefits for researchers and model engineers.

ADOxx puts a lot of emphasis on graphical visualization and interaction, i.e. model instances can be easily created with a graphical UI, where class instances and relationships can be instantiated and modified. The support for graphical editing is natively embedded in ADOxx, in contrast to other metamodeling platforms like EMF, where 3rd party graphic-based modeling environments have to be installed and configured in order to enable graphical model features.

ADOxx's strength is also attributed to the use of a rich-feature scripting language, namely *AdoScript*. Not only can constraints be defined by this language, but also procedures for model elements defined. The use of *AdoScript* can therefore extend the functionality of metamodel elements, which enables developers of metamodels to create domain-specific environments that are tailored to a pre-defined use case. The scripting language will be heavily used in the EMF to ADOxx transformation in section 5.3, where it enables the possibility to define behaviour not natively supported by the platform, but needed for the transformation purposes.

ADOxx is open-use but not open-source. This scientific work does, to some extent, rely on services made publicly available by the developers of ADOxx, like the *ADOxx ALL Public API* [adoc] mentioned in subsection 5.3.1, of which the source code cannot be viewed. This poses potential problems to validity of created artifacts, extensive trial and error phase in order to develop certain features and general feasibility concerns of the metamodel transformation.

2.2.2 Usage

In order to enable the ADOxx environment for metamodel engineers, it is needed to install the ADOxx standalone application. ADOxx requires a Microsoft SQL Server database instance, which is installed in this process, followed by the installation of the main applications needed for metamodel and model creation. The SQL Server instance is needed in order to store the metamodel and model data, which stems from the architecture and design choice of ADOxx. This is contrary to EMF, where the models and metamodels are stored in XML-based files and read in RAM just-in-time when they are used.

From a metamodel engineer perspective, the user has to "1) configure the specific metamodel by referring its concepts to the ADOxx meta-metamodel; 2) define a visualization for the concepts; 3) combine the concepts into logical chunks, i.e., ADOxx modeltypes; and 4) realize additional model processing functionality like model transformation or simulation." [BPB21].

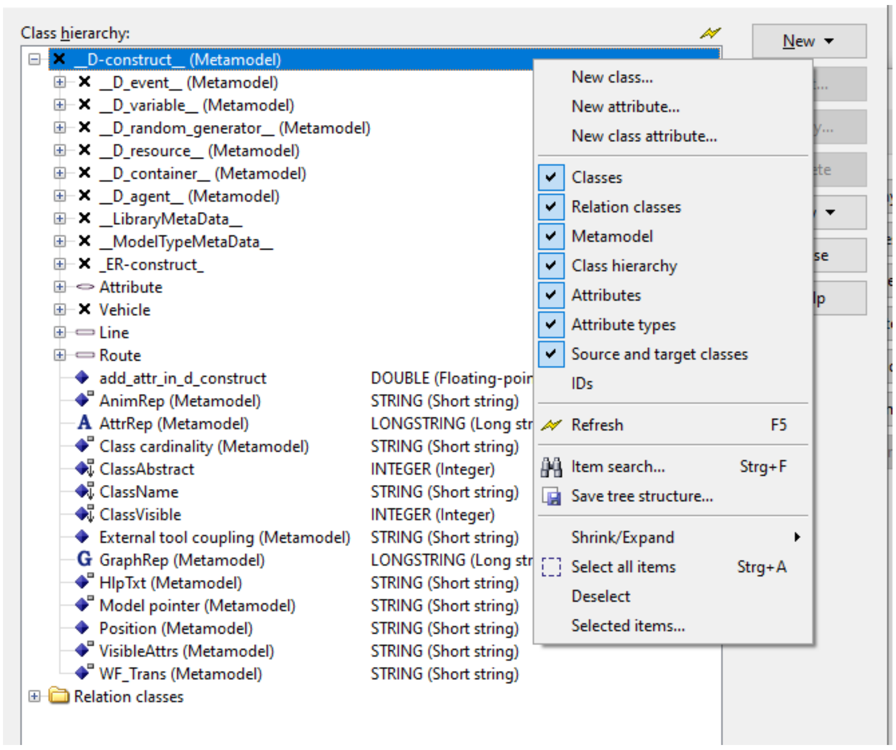
After the installation is finished, two applications are made available for the user:

- The ADOxx Development Toolkit
- The ADOxx Modelling Toolkit

The *ADOxx Development Toolkit* is the metamodeling tool of ADOxx. Through the development toolkit, users can be created and roles assigned as well as new metamodels instantiated, modified and exported for further purposes.

ADOxx supports two libraries per project, that can be used independently:

- The dynamic library
- The static library



Class hierarchy:

__D-construct__ (Metamodel)	
__D_event__ (Metamodel)	
__D_variable__ (Metamodel)	
__D_random_generator__ (Metamodel)	
__D_resource__ (Metamodel)	
__D_container__ (Metamodel)	
__D_agent__ (Metamodel)	
__LibraryMetaData__	
__ModelTypeMetaData__	
ER-construct	
Attribute	
Vehicle	
Line	
Route	
add_attr_in_d_construct	DOUBLE (Floating-poir
AnimRep (Metamodel)	STRING (Short string)
AttrRep (Metamodel)	LONGSTRING (Long str
Class cardinality (Metamodel)	STRING (Short string)
ClassAbstract	INTEGER (Integer)
ClassName	STRING (Short string)
ClassVisible	INTEGER (Integer)
External tool coupling (Metamodel)	STRING (Short string)
GraphRep (Metamodel)	LONGSTRING (Long str
HlpTxt (Metamodel)	STRING (Short string)
Model pointer (Metamodel)	STRING (Short string)
Position (Metamodel)	STRING (Short string)
VisibleAttrs (Metamodel)	STRING (Short string)
WF_Trans (Metamodel)	STRING (Short string)
Relation classes	

Figure 2.2: Sample ADOxx class hierarchy editor

The dynamic library is intended for the creation of metamodels in the domain of generic simulation (e.g. path analysis) and is mostly used by metamodel developers when creating any metamodel project. The static metamodel on the other hand provides a generic tree based structure, which can be helpful to more easily model static behaviour of a real world scenario (e.g. an organizational structure).

After selecting a specific library, the user can create and modify metamodel elements via the *Class Hierarchy Editor*. Figure 2.2 shows a sample ADOxx class hierarchy editor with a sample metamodel, as well as the various actions that can be performed on an element. Creating classes can be done via selecting a specific existing class, accessing the context menu and selecting the option `New Class...`. The class that was selected in this process automatically becomes the super class of the new class.

Attributes can be added to a class in a similar way, i.e. selecting a class, selecting the option `New Attribute...` or `New Class Attribute...`, specifying a name and an attribute type. `Relation Classes` can be created via selecting the folder `Relation Classes`, accessing the context menu and selecting the option `New relationclass`, where a name, as well a source and target class can be selected from a drop-down menu. Editing any element can be done via selecting the desired element, accessing the context menu and selecting the option `Edit...`, where ADOxx allows to modify various values, like name, datatype or abstraction flags, depending on the type of the selected element.

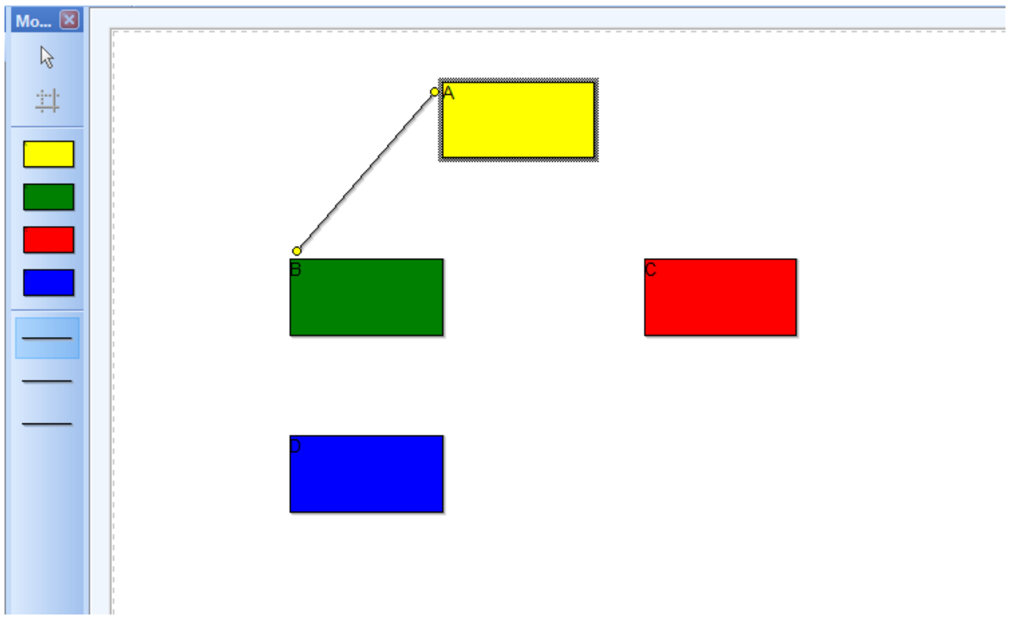


Figure 2.3: Sample ADOxx modelling toolkit palette and canvas

It is here, that various important attributes can be set, like the *AttrRep* attribute, that controls which attributes are actually editable in the model editor. But also the *ClassCardinality* attribute on `Classes`, which allows to specify min and max values of incoming and outgoing relationships. Worth to mention is also the attribute *AttrInterRefDomain* attribute, which is a distinct attribute of `Interref` instances that allows to specify the properties of such an element.

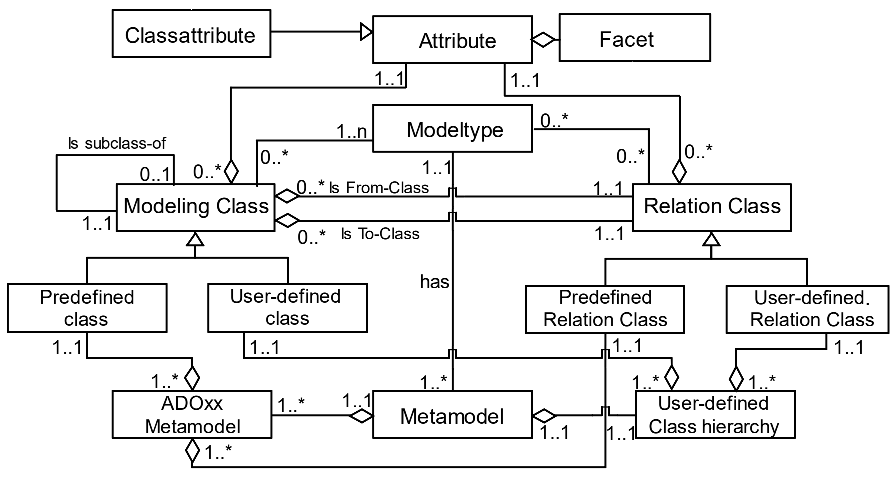
The *ADOxx Modelling Toolkit* on the other hand is used to create and modify models based on a pre-defined metamodel. After logging in to the Modelling Toolkit, the user can create a new model out of the Model Groups Explorer's context menu. The metamodels, out of which models can be defined, can be set in the user settings of the ADOxx Development Toolkit. Depending on which model or model groups were chosen, the user can select a new model type based on a certain metamodel.

Figure 2.3 shows a sample instance of the Modelling Toolkit, with the palette on the left, the canvas right next to it, and various classes instantiated inside of it. Adding class instances to the model can be preformed by selecting the appropriate element from the palette and clicking onto the white canvas next to it. The representation of palette items is defined through the *AttrGraphRep* attribute, that was modified in the ADOxx Development Toolkit on the corresponding class. The white canvas, where elements from the palette can be added, functions as a graphical representation of the metamodel, i.e. it corresponds to the underlying model, while providing graphical features that allow to easily interact with the model elements.

By double-clicking on a created class on the canvas, the Notebook pop-up of that particular class opens, where the class's attribute values can be modified. `Relation Classes` can be created by selecting the relation class from the palette, followed by a selection of two class instances on the canvas between which the relationship should be created. ADOxx gives hints as to which classes can be statically connected, i.e. if a source or target class matches the static specification defined in the ADOxx Development Toolkit for a particular relation class, the corresponding class receives a light-grey border when hovering over it. If the desired class is invalid for this type of relation class, no border appears and an error message shows up, stating an invalid-object error. Additionally, ADOxx allows to select the source and target class in opposite order when trying to create a relationship. This however does not resemble the inverse property that EMF posses, which will be described in section 4.3; rather the relationship is created internally in the pre-defined constellation of source and target class.

2.2.3 Metamodel

Figure 2.4 shows an excerpt of the ADOxx meta-metamodel. The emphasize in this meta-metamodel lies on the element `Metamodel`, i.e. the metamodels that can be created in the *ADOxx Development Toolkit* refer to this meta-metamodel element.

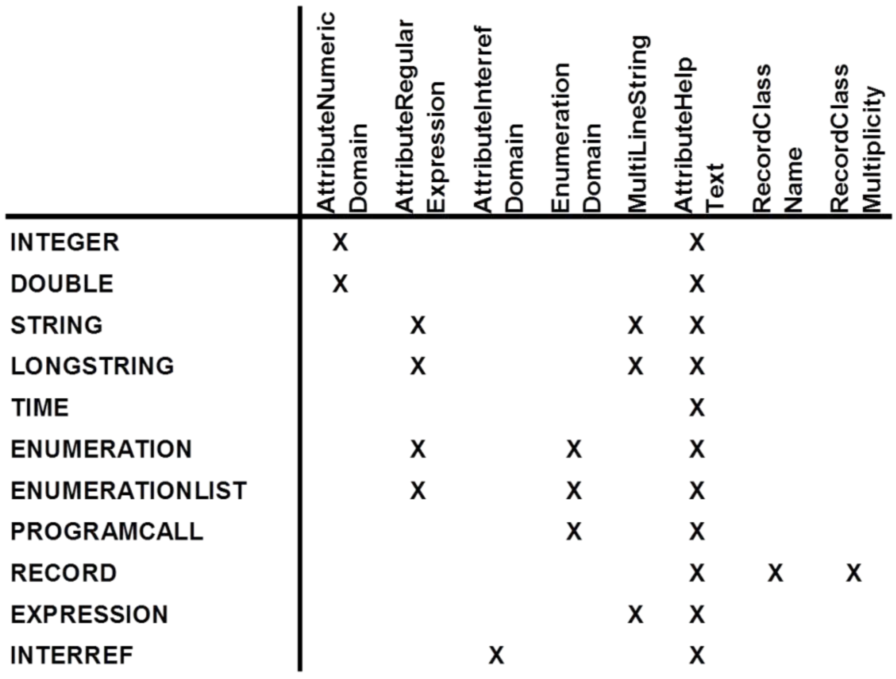


,

Figure 2.4: Excerpt of the ADOxx dynamic meta-metamodel [Bor18]

As can be seen, a `Metamodel` has exactly one `Modeltype`, which in turn consists of several `Modeling Classes` and `Relation Classes`, and is composed of the `ADOxx Metamodel`. Both `Modeling Classes` (which represent the first-class concept of *classes*) and `Relation Classes` (which represent the first-class concept of *relationships*), can contain several `Attributes`. An `Attribute` can also be of type `Classattribute`, where every class or subclass receives the exact same one value.[adof]

`Facets` in ADOxx are contained by `Attributes`. The `Facets` contain meta information for an attribute, e.g. a help text for a attribute, which is displayed to the user, or cardinality information, that enforce the instantiation behaviour of certain classes on the



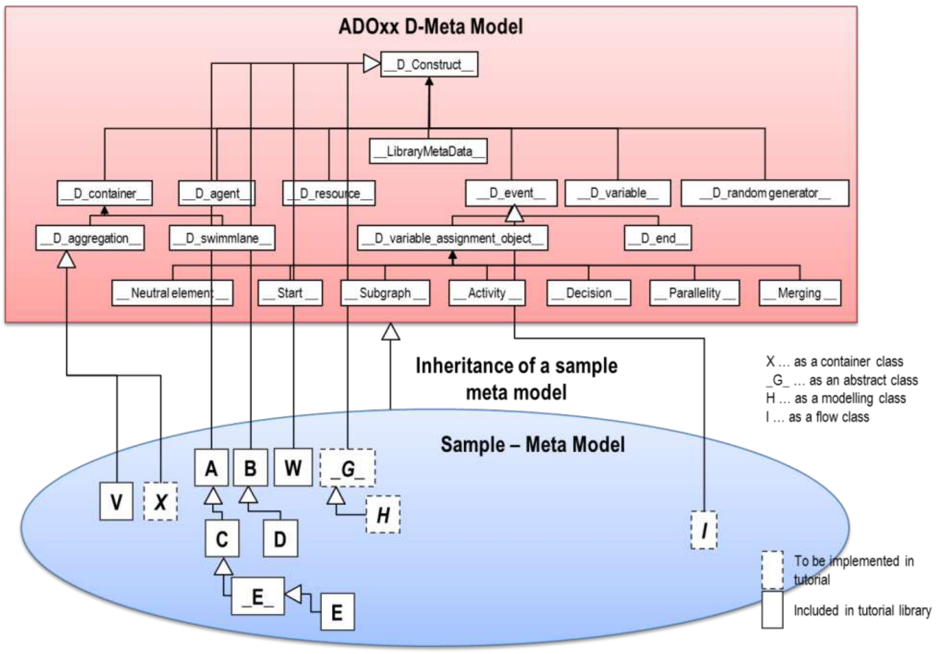
	AttributeNumeric Domain	AttributeRegular Expression	AttributeInterref Domain	Enumeration Domain	MultiLineString	AttributeHelp Text	RecordClass Name	RecordClass Multiplicity
INTEGER	X					X		
DOUBLE	X					X		
STRING		X			X	X		
LONGSTRING		X			X	X		
TIME						X		
ENUMERATION		X		X		X		
ENUMERATIONLIST		X		X		X		
PROGRAMCALL				X		X		
RECORD						X	X	X
EXPRESSION					X	X		
INTERREF			X			X		

Figure 2.5: Attribute Facets available per attribute type [adoe]

model plain. Different facets are available to different attributes of a certain type. An overview of the different facets and their corresponding attribute types can be seen in Figure 2.5. [adoe]

The `ADOxx metamodel` is a set of statically defined classes, that are part of every metamodel created within ADOxx. The team of ADOxx introduced certain name conventions and best practices in order to ease the development of metamodels and to introduce a clearer structure to the project: Double underscores before and after a class name mark this particular class as an abstract ADOxx metamodel class. Single underscores should be used by the metamodel developer to denote a user-created abstract class of the metamodel.

In Figure 2.6, an excerpt of the ADOxx dynamic metamodel can be seen in red, as well as a sample metamodel that contains elements referring to the metamodel element in blue. The class `__D-construct__` is the top class in the inheritance hierarchy, i.e. every other class that is either predefined by ADOxx or created by the user within the dynamic library, will inherit at some point from this specific class.



,

Figure 2.6: Relation of ADOxx dynamic metamodel and user-defined metamodel [adog]

2.3 EMF

2.3.1 Introduction

The Eclipse Modeling Framework (EMF) "can be considered as the Java-based realization of the Meta-Object Facility (MOF) standard" [BWA21]. It offers not only metamodel and model creation tools but also various features related to model transformation and model accessing. Brambilla et al. state in [BCWB17], that with EMF one can

1. Create and edit (meta)models
2. Use generation components for programmatic manipulation with (meta)models and tree-based modeling editors
3. Serialize and de-serialize (meta)models from/to XMI
4. Use additional modeling plug-ins

EMF provides a graphics-based tool for creating metamodels based on UML modeling techniques, which this work will further describe in subsection 2.3.2. EMF offers a tree-based editor within Eclipse to perform these tasks by creating models from a defined metamodel and modifying them.

EMF is strongly bound with the Integrated Development Environment (IDE) Eclipse, i.e., most tools that EMF provides can only be used within an installed instance of

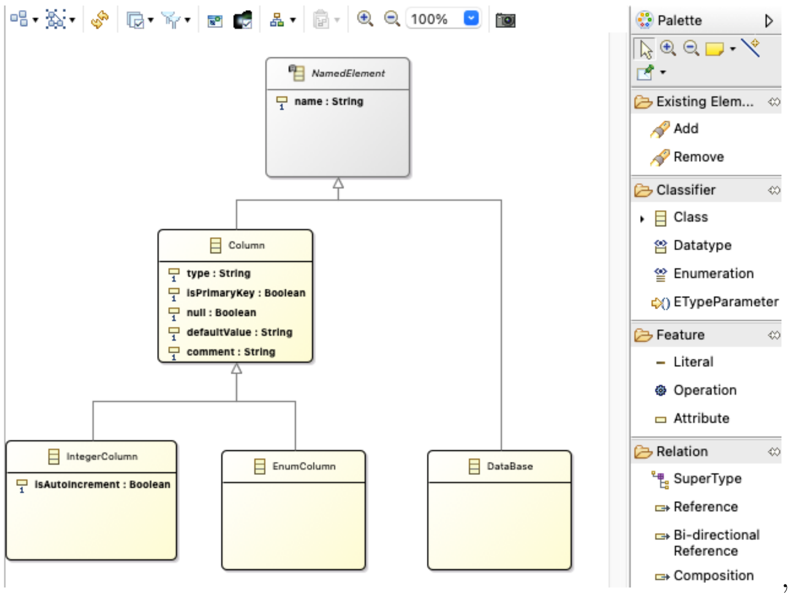


Figure 2.7: Sample EMF metamodel editor instance

Eclipse. However, as mentioned in the second item, programmatic solutions, like the Eclipse API [emfc], which will also be used in chapter 5, can provide a way to read and modify metamodel and model elements.

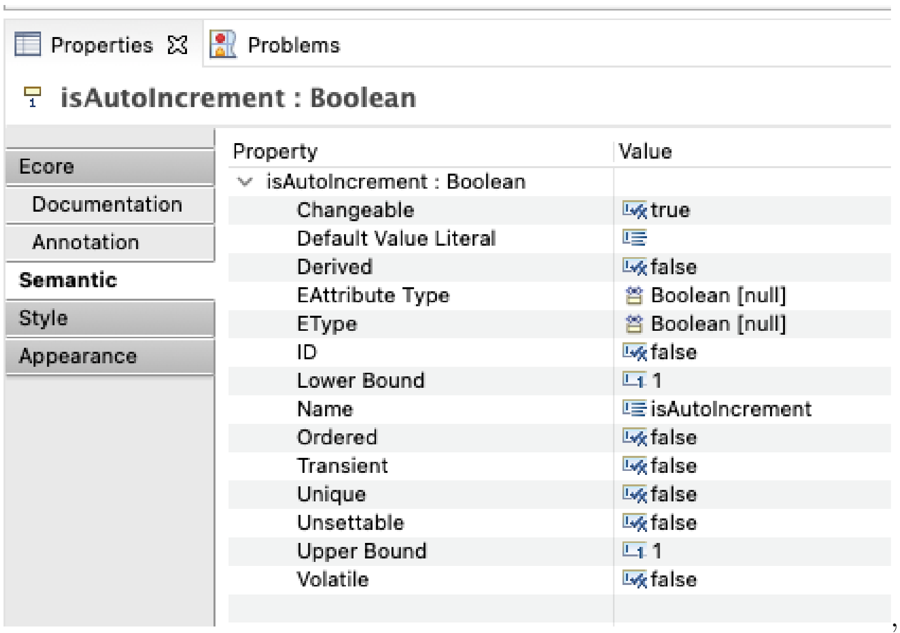
EMF uses XMI, which is based on XML [Gro15] to serialize and de-serialize metamodels and models. XMI is, on the one hand, easy to read and modify for developers, but since it is based on XML, it can be used by XML parsers to access and modify elements programmatically.

Lastly, one of EMF's characteristics is its strong support for plug-ins developed mainly by the EMF community. Prominent plug-ins are Sirius [emfd], which is a graphical modeling tool that allows, based on an EMF metamodel, to create an own graphic editor through which instances of models can be created. ATL [emfa] is another popular plug-in that allows performing model-to-model transformations with a tailored Domain Specific Language (DSL).

2.3.2 Usage

To enable metamodel creation and editing in EMF, one needs to download and install Eclipse as a prerequisite. The specific version with the EMF modeling tools can be downloaded; alternatively, a basic Eclipse version can be installed and the EMF modeling plugins added later via the plugin manager.

Figure 2.7 shows an example instance of a created project, which a metamodel engineer uses to create or edit metamodels.


Properties | Problems

isAutoIncrement : Boolean

Property	Value
isAutoIncrement : Boolean	
Changeable	true
Default Value Literal	
Derived	false
EAttribute Type	Boolean [null]
EType	Boolean [null]
ID	false
Lower Bound	1
Name	isAutoIncrement
Ordered	false
Transient	false
Unique	false
Unsettable	false
Upper Bound	1
Volatile	false

Figure 2.8: Sample properties section of an EMF metamodel element

The model engineer can add various elements of the metamodel on the canvas. The available options appear on the palette. The metamodel developer can initially only add items from the category `Classifier` to the canvas. Items from the category `Feature` can only be added inside those `Classifier` elements, while the user can only create objects from the category `Relation` between them. The EMF metamodel editor generally hints which elements the user can interact with within a specific context, i.e., certain non-allowed symbols appear on the cursor if the user hovers over an invalid element.

When clicking on an item, like a class or an attribute, the `Properties` section of the editor changes accordingly, giving the developer an interface through which he can modify certain meta-information of an element. Figure 2.8 shows an example of the different flags and other meta-information that can be configured for an `Attribute`.

There are different ways to create a model out of an existing metamodel in EMF. Creating a dynamic instance will be described here, although creating an own Eclipse Runtime from an existing metamodel and instantiating models from there is also possible.

When the metamodel development finishes, the developer can create a new dynamic instance from the context menu of a metamodel element. After assigning a name, a new file with the ending `.xmi`, EMF's serialization format, is created in the Eclipse project. As can be seen in figure Figure 2.9, the EMF model editor gives the model editors the chance to create elements based on a previously created metamodel. Accessing the context menu of a certain item allows the developer to create new instances of model

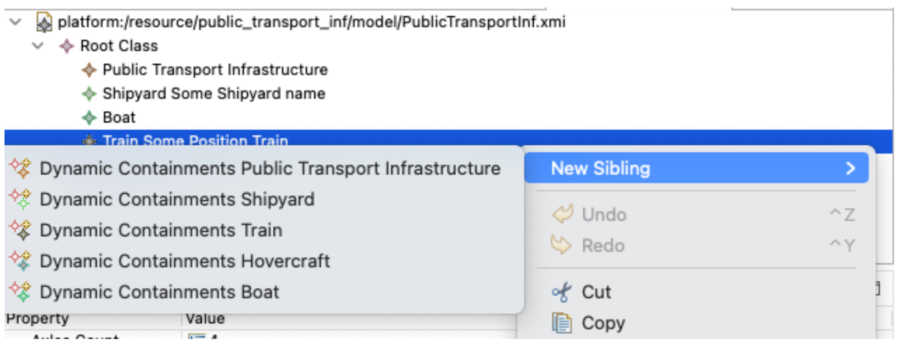

platform:/resource/public_transport_inf/model/PublicTransportInf.xmi
Root Class
Public Transport Infrastructure
Shipyard Some Shipyard name
Boat
Train Some Position Train
Dynamic Containments Public Transport Infrastructure
Dynamic Containments Shipyard
Dynamic Containments Train
Dynamic Containments Hovercraft
Dynamic Containments Boat
New Sibling
Undo ^Z
Redo ^Y
Cut
Copy
Property
Value

Figure 2.9: Creating and editing a sample EMF model

elements, either as children (i.e., contained within an element in this tree structure, like `Root Class` and `Public Transport Infrastructure`, but also as siblings (i.e., on the same hierarchy level) like `Shipyard` and `Boat`, based on the defined structure of the metamodel.

Finally, as with the metamodel elements, a model element might have its metadata (i.e., its defined attributes in the metamodel) modified through a similar interface as in Figure 2.8.

2.3.3 Metamodel

Ecore is the name of the meta-metamodel of EMF. Classes in Ecore have the prefix `E`, followed by the name of the first-class concept they resemble. `EClasses` therefore resemble the concept of class, `EEnum` resemble the concept of Enums and so forth.

As can be seen in the excerpt of the Ecore meta-metamodel in Figure 2.10, every class inherits from the abstract class `ENamedElement`, which ensures that every element of the generated metamodel has a name assigned to it.

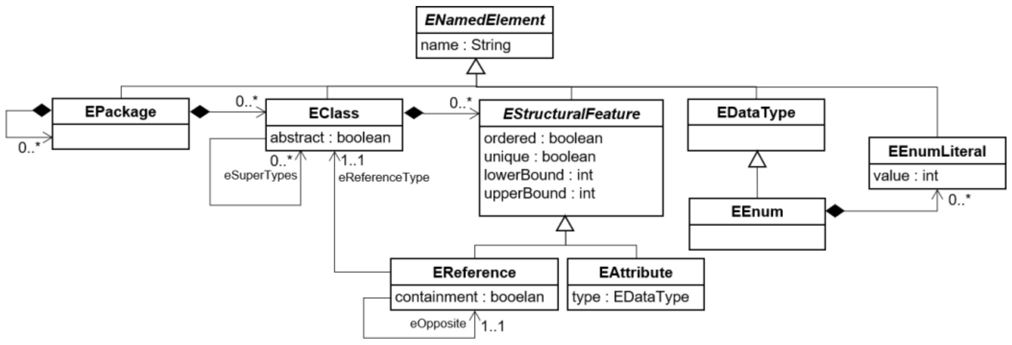


Figure 2.10: Excerpt of the Ecore meta-metamodel[BWA21]

The `EPackage` element resembles the root element of every metamodel. `EPackages` can be nested (as seen by the self-composition), but also contain instances of `EClass`, which in turn can be either abstract or instantiable and have a super type reference to other `EClasses` or data types, of which `EEnums` are a part of. `EEnums` are also composed of an arbitrary number of `EEnumLiterals`, which resemble the different values an enum can possibly have.

`EStructuralFeatures` are classes that contain specific attributes like *ordered*, *unique*, *lowerBound* and *upperBound*. `EAttributes` inherit from this class, while also providing the attribute *type*, which can be set to an arbitrary type that can be part of the Ecore metamodel (like `EEnums`) or regular Java classes.

`EReferences` are also a sub type of `EStructuralFeature`, and define additionally the Boolean flag *containment*, which marks if this particular relationship resembles a composition. If a class has this ability enabled (i.e. it is a compositum), every composed instance can only be created through an instance of this class. If an instance of compositum is deleted on model level, all composite elements will then be deleted as well.

What is different from other metamodeling platforms is that the Ecore's meta-meta model has its relationships contained inside an `EClass`, as opposed to ADOxx, where a relationship is represented as a separate object on the root metamodel level, together with other class definitions.

2.4 Interoperability

Interoperability in the context of MDE describes the ability to interchange metamodels and models between two selected tools [Ker08]. This definition implies that a model engineer can use a metamodel of one platform in the other platform and vice versa.

Such a procedure would yield several benefits: For one, the developer of a metamodel can switch to a different tool that is either more familiar to him or her or possesses improved features that make the development of a metamodel faster and easier. Several people or teams can work together on a common metamodel while using different metamodeling tools. This benefit is advantageous when two various parties are familiar with one metamodel editor but not the other. Following this procedure could reduce costs within the overall project since no team has to be trained on the other platform, enabling the development of robust metamodels. Model engineers and domain experts could then use the best tool for a particular job.

In [KM06], Kühn and Murzek state, that interoperability in MDE involves two dimensions:

1. Information Heterogeneity and
2. System Heterogeneity

Information Heterogeneity describes the level of heterogeneity of metamodel information, both semantic and syntactic when comparing the metamodels of the source and target platform. Semantic heterogeneity is measured by the degree of meaning of a metamodel being equivalent, while syntactic heterogeneity is evaluated by the manifested structure of the source-platform metamodel conforming to the structure of the target-platform metamodel.

Information Heterogeneity is essential for interoperability since the information must be consistent for both the source and the target platform metamodel. Semantic heterogeneity is considered more important than syntactic heterogeneity. The meaning of the metamodel being consistent in both platforms can help create a shared experience in both platforms, therefore strongly enabling interoperability. Syntactic heterogeneity, while less important, is not negligible, though, since static behavior, e.g., a much bigger file size of metamodels in the target platform, can create a bad experience due to worse performance, lowering the value of a particular interoperability solution.

System Heterogeneity describes the layer of different metamodeling platforms' capabilities, such as the distinct creation, representation, and persistence mechanisms of metamodels and models. System Heterogeneity is important when proposing an interoperability solution, as different features and procedures present in either platform can reveal considerations when choosing a particular approach. A high degree of system heterogeneity between two platforms could ease the interoperability due to lots of commonalities between them, while a lower degree might even impose barriers, like certain features not being present in the other platform, which makes interoperability to some degree infeasible.

One way to achieve interoperability is to take a metamodel of one platform and transform it into the metamodel of another platform by applying specific rules derived from the characteristics of either platform, followed by a concluding comparison of these characteristics. This scientific work uses this approach to enable the interoperability of the metamodeling platforms ADOxx and EMF.

CHAPTER 3

Related Work

Some past research, which is related to this work, has already been conducted. This chapter tries to summarize those findings, emphasizing this scientific work's content, which leads to a better understanding of the state-of-the-art and the different techniques that are used throughout this thesis.

Some implemented approaches that enable interoperability between platforms other than ADOxx and EMF already exist. These include the work of [BCC$^+$10], where the authors present a method for metamodel transformation and provide an example. The work includes the derivation of transformation rules based on the meta-meta models of the respective metamodel environment and (semi)automatic application of these rules on existing metamodels. The authors provide an example transformation where a bridge between EMF and the Microsoft SQL server modeling tools is successfully created.

This thesis is heavily inspired by the work of [Ker16], who created (meta)model bridges between the metamodeling environments EMF and ARIS, MetaEdit+ and Microsoft Visio, respectively. The author uses M3 level-based bridges per two metamodeling environments, providing a foundation for all related model transformations. Transformations on the M3 level represent a transformation on the meta-meta model level. The author analyzes the differences and similarities of both meta-meta models and derives mapping rules that define how the transformation procedure maps one item or structure from a source environment to the target environment. Besides the metamodel transformation (i.e., M2 level transformations), the author also realizes model instance transformations (i.e., M1 level transformations) between two environments based on mapping definitions that are derived from the metamodel transformation.

Additionally, the author presents an approach that shows that it is not needed to create $N \cdot (N-1)$ bridges to create full interoperability for a set of metamodel environments of size N. It is sufficient to create only $N-1$ bidirectional bridges between a central metamodel environment (EMF in this case) and all other environments and use this central entity as an intermediary instance for all other transformations. Assuming that bidirectional bridges exist between an arbitrary metamodel environment A and the central metamodel environment C and the metamodel environments B and C, one does not need to create a new transformation from $A \rightarrow B$. Instead, one can use the existing transformation from A to the central environment C and from C to the target environment B ($A \rightarrow C \rightarrow B$).

Additional work is present in the broader term of metamodeling interoperability. In this research area, interoperability is not bound to a concrete metamodeling platform but to the transformation of the level of technical spaces. Those technical spaces include Modelware (i.e., model languages and tools) and Grammarware (i.e., textual-based languages and tools). These terms contain a broader set of meta-meta models, thus not being bound to any specific metamodel as presented in the previous works.

A bridge between those two technical spaces has been conducted in the work of [WK05]. In this work, the authors analyzed a more general approach to metamodel and model transformation based on the EBNF of the specific language. The transformation is based on the M3 level, where the derivation of rules and the transformation's execution is performed automatically.

Additional findings on this more general approach appear in the work of [NBM+15], where the authors created a bridge between Modelware and XMLware. XMLware comprises the fields of XML processing and the XML representation. A similar bridge between Modelware and JSONware is described in the work of [CGB+21]. As with XMLWare, the authors analyze the processing and representation of schemata, but in the context of the meta-language JSON. With the findings of these two interoperability analyses, the benefits of XML and JSON documents, i.e., well-structuredness or readability, can be combined with model-driven aspects. The results could be model instances that a software developer created out of a given XML or JSON definition or schema documents or concrete schemata corresponding to a provided model of a concrete metamodeling platform.

In [KBJK03], the authors described the concept of metamodel integration patterns by introducing the method of Enterprise Model Integration (EMI). Instead of defining concrete mappings for elements from one environment to the other, patterns like the aggregation or the reference pattern are defined, which can be used to generalize metamodel integration using object-oriented meta-modeling concepts.

Finally, a research group researched the field of method chunks. "A method chunk is an autonomous, cohesive and coherent part of a method providing guidelines and related concepts to support the realisation of some specific system engineering activity"[RBKJ06]. Solutions to various interoperability problems are connected to method chunks, accompanied by guidelines and possible examples that help the model engineer create interoperability by applying them to existing metamodels.

As could be seen, there is a multitude of work present on interoperability of framework-combinations other than ADOxx and EMF and several works on more general interoperability approaches in the context of metamodeling. But as of now, there exists no scientific work that establishes a bridge between ADOxx and EMF. While an analysis of language features of ADOxx has been made in the work of [FRK06], and several studies of EMF have been made in other works, e.g., in [Ker16], no work has thus combined the knowledge to create a bridge and enable interoperability between these two metamodeling platforms.

CHAPTER 4

Comparative analysis of ADOxx and EMF

In the previous chapters 2.2 and 2.3, the foundations of the two meta-modeling platforms ADOxx and EMF have been described. The next step is to compare the two platforms in the context of their meta-meta model features. This comparison gains insight into the abstract syntax of each platform, which helps to implement the metamodel bridges at a later step.

The comparison is divided into different groups, i.e., *Core Modeling Concepts, Classes, Relationships, Attributes, Inheritance, Grouping* and *Constraint language*. Each group then lists the different concrete feature implementations on each platform.

This comparison helps to comprehend the differences and similarities between the two platforms. Their results will also build the basis for the transformation rules that will be used in the chapter Metamodel Transformation, thus making them an essential part of this thesis. The Table 4.1 summarizes the findings of this comparison in an illustrative way. The similarities and differences of these meta-meta model features will be described in detail per category in the upcoming sections.

Table 4.1: Comparison of M3 Level features of ADOxx and Ecore [BWA21]

Criteria	**ADOxx**	**Ecore**
Core Modeling Concepts		
Class	Class	EClass
Relationship	Relation Class	EReference
Attribute	Attribute/ Class Attribute	EAttribute
Classes		
Abstract Classes	✓	✓
User-defined root element	✗[1]	✓
Relationships		
Arity	binary[2]	binary
Inverse	✗[3]	✓
Composition	✗[3] (only visual)	✓
Multiplicity	✓	✓
Endpoints	Class	EClass
Unique Names	✓(per Metamodel)	✓(per Class)
Link to Model	✓	✗
Attributes		
Applicable to	Class/Relation Class	EClass
Multiplicity	single-/multi-valued	single-/multi-valued
Unique	✓	✓
Ordered	✗[3]	✓
Default Value	✓	✓
Custom Data Type	✓[4]	✓
Inheritance		
Single/Multiple	single	multiple
Instantiation	single	single
Class Inheritance	✓	✓
Relationship Inheritance	✗	✗
Grouping	ModelTypes	EPackage
Constraint Language	AdoScript	OCL

[1] every class in ADOxx inherits from a predefined abstract class
[2] n-ary with *Interref* [3] realization via AdoScript possible [4] via Record Classes

4.1 Core Modeling Concepts

Core modeling concepts are basic features that describe the way in which information can be persisted and relationships defined.

Class Classes in ADOxx are represented by the `Class` feature. The equivalent item on the Ecore site is `EClass`. Classes in both platforms resemble the MOF element of class.

Relationship `Relation classes` are used on the ADOxx side to determine a relationship between two classes. Ecore uses the `EReference` feature to represent a relationship between 2 classes. Notably, ADOxx implements here the `Relation classes` as standalone features on the top level of the metamodel. In contrast, `EReferences` are contained within the class object where the relationship originates from.

Attribute Attributes in ADOxx are represented by either the `Attribute` feature or the `Class Attribute` feature. As described in the ADOxx documentation, the difference is that "Class attributes receive one value for every class. Instance attributes receive one value of each instance or relation."[adoa]. `Class Attributes` in ADOxx represent static attributes in programming languages like Java, where only one value per class can be defined. A class attribute is final, i.e., it is only definable within the ADOxx Development Toolkit and not changed in model instances in the ADOxx Modelling Toolkit.

The equivalent feature to an attribute in Ecore is the `EAttribute` type. Ecore does not support the feature of static attributes. This work describes concrete data types and features both in ADOxx and EMF in the Attributes section.

4.2 Classes

Classes in metamodeling are a central feature. They support the ability to hold certain information about themselves, like names, attributes, and relations. References can only be defined between and attributes can only be defined inside them, further proving their relevance in this domain.

When comparing the two metamodeling platforms ADOxx and EMF, certain features are implemented differently in either:

Abstract Classes In both ADOxx and EMF, an abstract flag can be selected, denoting that specific class as abstract. Once a class is abstract, the metamodel environment does not allow creating an instance of that particular class on the model level. Only non-abstract sub classes of an abstract class are instantiable. This behavior applies to both ADOxx and Ecore.

User-defined root element In ADOxx, there is no need to have a user-defined root class to create a concrete model object on the model level. Every class inherits from a predefined class, i.e., the `RootClass`. This class is implicit and cannot be changed or omitted. On the model-editor site, the model engineer can fill the canvas with various instances of `RootClass`. Since all classes inherit from this class, any class is addable to the model without constraints.

On the other hand, Ecore forces the user to create a root element, out of which he or she can create all other elements on the model level. This root element has to be selected on metamodel creation. An instance of any other class can only be added to the model when this root class has a *contains*- relationship to that other class.

4.3 Relationships

Relationships in metamodeling define the connectedness between two components. In most cases, the connecting components are classes, but they can also be of a different type, as described below. Relationships can also possess different attributes and support different behavior, where a description for both platforms follows in this section.

Arity Both in ADOxx and Ecore, relationships (`Relation Class` and `EReference`) are binary, meaning that the degree of relatable objects that participate in a relationship is two.

ADOxx additionally allows to reference multiple other elements from one class with the help of `Interrefs`. An attribute with this datatype can be defined inside a class, which can have links to several other classes, all with a possible different cardinality. This way, one is able to define custom n-ary relationships with different cardinalities in ADOxx.

Inverse In Ecore, a relationship can be bidirectional by checking a flag in the relation's properties. Therefore, one can define an inverse of a relationship, which has the opposite source and target of the original relationship. In ADOxx, such a feature is not present. The metamodel engineer can only define a relationship in one way. An inverse relationship would have to be defined explicitly.

Composition In ADOxx, compositions are not a built-in feature of the platform. Thus one cannot create Compositums and Composed Instances and perform a certain action, e.g., when a Compositum is deleted, the composed instances are also deleted. However, it is possible to create such behavior in ADOxx via AdoScript. This will be described in more detail in section 5.2. Instances of `__D_Container__` however can be used to provide visual, but not semantic compositions, i.e., the container instance can hold several sub-class instances and group them visually in the model editor.

In Ecore, the feature of compositions is supported natively by the platform. Therefore creating composition features within the models and metamodels does not require additional code as with ADOxx.

Multiplicity Both in ADOxx and Ecore, one can define the multiplicity of a relationship, i.e., determine the upper and lower limits of instantiable objects throughout a model that are part of this particular relationship. While Ecore allows setting multiplicity-values of a certain relationship for two parameters, namely `Lower Bound` and `Upper Bound`, ADOxx comes with the possibility of modifying four parameters per involved class, i.e., `min-outgoing, max-outgoing, min-incoming` and `max-incoming`, totaling in 8 different alterable parameters.

Endpoints ADOxx allows to define relationships only on the core modeling concept of `Class`. Ecore also only allows to define relationships on the core modeling concept of class, this being `EClass`.

Unique Names Relationships in ADOxx are structures defined on the highest metamodel layer. Since they are treated similarly to classes, one cannot duplicate a class and use its exact name again in the metamodel since they must be distinguishable. Therefore ADOxx does not support defining multiple relationships with the same name within a metamodel.

Ecore, on the other hand, does not have this limitation since relationships are defined within every source object rather than on the metamodel's top layer. While it is still impossible to define two relationships with the same name within a source class, it is possible to use that same relationship name in other classes of the same metamodel.

Default Value Both ADOxx and Ecore allow adding a default value to relationships, which the metamodel engineer can use to store additional information as text for that relationship.

Link to model ADOxx allows creating links to a model via `Interrefs`. Their usage comprises not only referencing another class (as in an internal relationship between two objects of the same metamodel) but also referencing a whole other metamodel (which can be achieved by using the `mref` keyword in the `Interref` definition).

4.4 Attributes

Metamodeling platforms use attributes to store additional information on certain metamodel objects. Usually, attributes can be added to class elements, allowing the model engineer to edit specific additional information of a class. Attributes can possess different properties and capabilities, which is compared for ADOxx and EMF in this section,

Applicable to Both ADOxx and EMF possess attributes on classes and relationships. Attributes are however, in ADOxx, addable to a `Class` and `Relation Class`.

In Ecore, the metamodel developer can only add attributes to an EClass. While there is a possibility to change values of pre-defined attributes in `EReferences`, there is no possibility to add new ones.

Multiplicity Both in ADOxx and Ecore, it is possible to define attributes that function both as single-valued attributes (e.g. of type integer, holding a numerical value) and multi-valued attributes (e.g., a multi-selection enum). Multi-valued attributes in ADOxx are represented by the datatype `ENUMLIST`, which allows selecting multiple values of a pre-defined `ENUM`. In Ecore it is possible to define a specific upper bound of an `EEnum`, allowing to select multiple values for one attribute, thus making Ecore's multiplicity multi-valued as well.

Unique Both in ADOxx and Ecore, it is impossible to define two attributes with the same name inside one class, even when defining different attribute types. Thus, attributes are unique on both platforms.

Ordered In EMF, it is possible to enable the *ordered* flag on an attribute, which will cause the affecting attribute values to be in a natural order (e.g., ascending by string characters or numbers). ADOxx does not support this behavior natively, but its realization is feasible when writing AdoScript that reproduces this logic.

Default Value Both metamodel platforms support the possibility to define default values for any attribute type.

Custom Data Type In ADOxx, it is possible to define custom data types via `Record Classes`. One can select the entire library and create a `Record Class`, which can be then be used by any attribute which is part of the static or the dynamic library by assigning the attribute data type to the newly created `Record Class`.

In Ecore, it is also possible to define custom data types on the metamodel level within the metamodel editor. Once defined, the metamodel developer can assign the custom data type to any attribute within the same or other referenced metamodel.

4.5 Inheritance

Inheritance is a key feature when it comes to metamodeling. Through inheritance, the metamodel developer can define the same behavior and properties in one central place for the class itself and all its subclasses, rather than defining explicit behavior for every class in question on every change. There are differences in essential features for inheritance patterns, but also some similarities in ADOxx and EMF:

Single/Multiple In ADOxx, it is only possible to inherit from one single other class, thus only supporting single inheritance on the metamodel level. In Ecore, it is possible to inherit from an arbitrary number of classes. Thus EMF supports multi-inheritance. This multi-inheritance is one of the key differences between the meta-metamodeling concepts of ADOxx and EMF, which needs additional procedures to achieve semantic correctness on ADOxx when performing the transformation from EMF to ADOxx. This scientific work mentions this procedure in more detail in section 5.3.

Instantiation Instantiation in this context describes the multiplicity of instantiable classes in a multi-inheritance relationship. It, therefore, describes the assignment of a sub class to a super class type.

In ADOxx, the instantiation is trivially single since only single inheritance is supported. Therefore every instance of a sub class can only be attached to a super class of one type. In Ecore, a sub class is also always connected to one type of super class, although multiple inheritance is supported in this environment, which could support multi-instantiation patterns.

Class Inheritance Both ADOxx and Ecore allow inheriting classes from one another. The hierarchical depth for both platforms is unlimited.

Relationship Inheritance In both metamodeling platforms, it is impossible to inherit a relationship from one another, thus making relationships final in terms of extendability.

4.6 Grouping

Grouping describes the term of bundling multiple metamodel elements inside a container, creating benefits like applying modifications or relations to the container, which are implicitly assigned to all contained elements, rather than having to define that same behavior for each contained element individually. Grouping also helps to separate concerns, e.g., where a developer can create different Grouping objects for different sub-domains within a domain, yielding an order and clear structure to the metamodel.

Grouping In ADOxx, the metamodel developer can achieve grouping by using the data type `ModelTypes`. These `ModelTypes` can be created by accessing the library attributes of a certain Library, assigning a name to the `ModelType`, and including the various classes and relationships that should be part of this `ModelType`.

Ecore offers the data type `EPackage` to group any metamodel elements within a container. The metamodel engineer can create these package objects either on the root metamodel level or inside any other instance of `EPackage`. `EPackages` are an important feature in Ecore when it comes to importing them inside another metamodel or when using the Ecore API, which is an essential part of the model and metamodel transformation described in chapters chapter 5.

4.7 Constraint Language

Constraint languages are essential for defining additional behavior on metamodel and model elements. This additional behavior can be either constraints (e.g., limiting a certain number of instantiable classes when a condition is met) or automated features embracing specific behavior (e.g., deleting composed instances of a composition; a vital component of the metamodel transformation which is described in section 5.2).

Constraint Language In ADOxx, the constraint language for defining custom behavior is called `AdoScript`. AdoScript is a rich DSL that allows defining any type of behavior on model level, thus also functioning as a constraint language since the metamodel engineer can easily define constraints through it. [adoh]

Ecore uses the Object Constraint Language (`OCL`)[BG14], which allows the metamodel editor to define additional constraints for metamodel objects, which in turn, the metamodel framework applies on the model level. EMF, which is the most popular metamodeling framework in the domain of Model-Driven Engineering, has adopted OCL, which is also one of the most widely used constraint languages for model engineering: "OCL has become a key component of any model-driven engineering (MDE) technique as the default language for expressing all kinds of (meta)model query, manipulation and specification requirements."[CG12].

CHAPTER 5

Metamodel transformation

The metamodel transformation between ADOxx and EMF is the crucial feature to enable interoperability of the two platforms. By creating such a transformation, one can take a metamodel file from a source platform and let the transformation project produce a valid transformed metamodel file for the target platform.

The generated artifacts of a metamodel transformation are the transformed metamodel file, which metamodel engineers can later import to the target environment. Therefore, the transformation project consists of two unidirectional transformations, one for transforming ADOxx metamodels to EMF metamodels and one for transforming EMF metamodels to ADOxx metamodels. A description of both of these procedures follows in more detail in section 5.2 and section 5.3.

5.1 Transformation overview

In this section, the overall transformation in both directions is described, to give a holistic picture of the two transformations and show their interconnectedness. Figure 5.1 shows a technical overview of both transformations. On the left side, every aspect of the ADOxx ecosystem is pictured, on the right side every aspect of EMF. The middle part comprises all the transformation relevant features.

The top section of the left and right boxes represents the respective platforms' meta-meta models. They are an essential asset in this transformation, as the transformation procedure uses their APIs to create specific metamodels elements, which conform to this meta-meta model. The Mapping-based Transformation Specification, which is used to map features from one platform to another, uses both of these meta-meta models.

On the left and the right side, one level below, reside the metamodels of either platform, which are instances of the respective meta-meta model. One side always resembles the input of a transformation; the other side resembles the output.

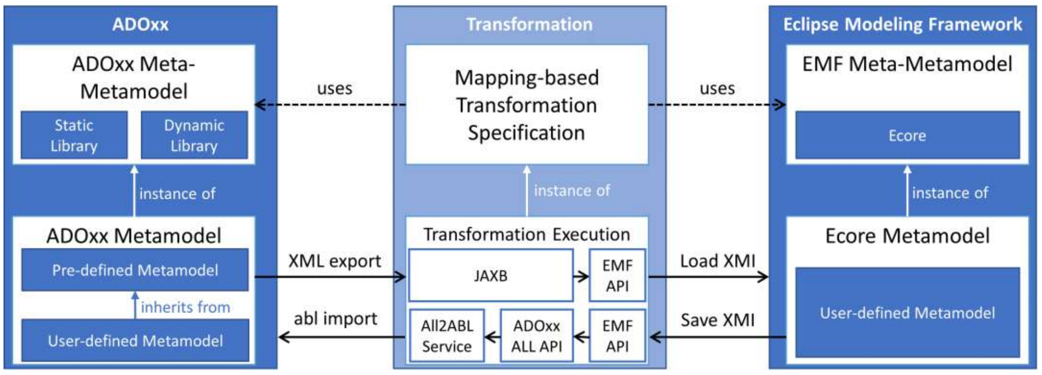


Figure 5.1: Technological view on the two unidirectional transformations [BWA21]

The ADOxx metamodel consists of a pre-defined metamodel and a user-defined metamodel, inheriting from the previous one. The pre-defined metamodel contains all elements which are part of the ADOxx dynamic library. In contrast, the user-defined metamodel includes all elements that a user added to a certain metamodel. Ecore does only contain one user-defined metamodel, i.e., it does not have a pre-defined metamodel that needs to be included in every metamodel instance.

Looking at the transformation direction from ADOxx to EMF, the transformation procedure first takes an XML export file from an ADOxx metamodel. It then transforms the file contents to a Java model via JAXB, which has the advantage of accessing the different features of the metamodel in Java in a programmatic way. With the help of the *EMF API*, the procedure creates an `.ecore` file that contains all mapped elements in an xmi-serialized format. A user can then import this file to EMF.

Looking at the other direction, and `.ecore` file, which represents a certain metamodel, is taken and read into a Java program via the *EMF API* [emfc]. With the help of the *ADOxx ALL Public API* [adoc], it is possible to create a complete ADOxx metamodel `.all` file, which includes both the pre-defined metamodel and the user-defined metamodel. Following this step, the procedure calls the *ALL2ABL Web Service* [adod] to create an `.abl` file from a given `.all` file. A metamodel engineer can import only this file to ADOxx via the ADOxx Development Toolkit. The transformation procedure eventually performs this task as its last step.

5.2 ADOxx to EMF

This section contains the description of the transformation of an ADOxx metamodel file to an Ecore metamodel file by referencing the used technologies, followed by an explanation of the actual procedure.

5.2.1 Used technology

Various reasons led to the use of the programming language Java for implementing the transformation program from ADOxx to EMF. One of the reasons is the great familiarity with the Java language due to long-term experience in previous projects. The capabilities of Java are clear, and we soon discovered that Java would be powerful enough to enable a transformation correctly and efficiently.

One key advantage of Java is the implementation of the software engineering concept of Object-Oriented Programming (OOP), which is suited for this project when it comes to accessing and transforming the various components by their semantic belonging. Java is also beneficial in that it supports different libraries that makes the transformation easier from a programmatic point of view.

The first library used in this procedure is `JAXB`. `JAXB` is a popular library that enables developers to read an XML file and transform it into a Java-based object structure (i.e., unmarshalling). Since the ADOxx metamodel export is an XML file, this library is well suited to read in the file and instantiate a Java object containing all the relevant information. A prerequisite is to create a Java model with different Java classes that would incorporate all the various elements, containments, and correct assigning of values and attributes from an ADOxx XML metamodel file. Numerous transformation projects are available online that transform an exemplar XML input to Plain Old Java Objects (POJOs). For this work, the tool JSON2CSharp [jso] is used, which contains an XML to Java transformation routine that created the corresponding Java files with a distinct naming pattern.

During the development process we discovered, that some information are not correctly mapped (e.g. in some occasions, attributes of type `Double` are not correctly interpreted by the XML-to-POJO-tool, leading to falsely assigning the type `Integer` to them). Those tool errors were resolved manually during the implementation process.

After adding this structure to the project, one is able to perform the unmarshalling by using the following commands as shown in Code Example 5.1

```
JAXBContext jaxbContext = JAXBContext.newInstance(Library.class);
Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
Library library = (Library) jaxbUnmarshaller.unmarshal(file);
```

Code Example 5.1: JAXB unmarshalling of ADOxx metamodel XML file

5.2.2 Transformation procedure

The overall procedure for transforming the metamodels consists of several procedural steps. An overview is shown in algorithm 5.1. Each step is described in more detail further below in this section.

One thing worth noting, which applies to all structures that the transformation procedure creates in this transformation, is naming convention issues. EMF follows certain naming conventions directly adapted from Java's naming conventions. This phenomenon stems from the fact that it is possible to create Java files from a given metamodel, which must be valid and consistent. For example, underscores are not allowed in certain places of

Algorithm 5.1: Overall procedural transformation from ADOxx to EMF

```
   Input: (adoxxLibrary)
1  $initPackages()$
2  $initHelperClasses()$
3  $addClassesToPackages()$
4  $addSuperTypesToClasses()$
5  $addAttributesToClasses()$
6  $addRelationshipsToClasses()$
7  $addBasicContainments()$
8  $resolveDuplicateNames()$
9  $outputToEcoreFile()$
10 $outputMappingFile()$
```

attributes; classes are not allowed to start with a numerical literal. Before adding a certain structure, a procedure is performed that takes a possible name as input and transforms it into a valid name. This procedure can be seen in algorithm 5.2.

Algorithm 5.2: Name transformation algorithm to create valid names in EMF

```
   Input: (nameCandidate)
1  foreach $literal \in nameCandidate$ do
2      if $isIllegalCharacter(literal)$ then
3          $nameCandidate \leftarrow replaceCharacterByLegalOne(nameCandidate, literal)$
4      end
5  end
6  if $nameCandidate.length == 0$ then
7      $nameCandidate \leftarrow' M'$
8  end
9  else if $nameCandidate.startsWithNumber()$ then
10     $nameCandidate \leftarrow' M\_' + nameCandidate$
```

Init Packages The first step of the transformation is to create 3 packages. those are:

1. ADOxx metamodel package
2. Metamodel dynamic package
3. Metamodel static package

In total, the transformation procedure creates three `EPackages` due to 1) a separation of concerns between the ADOxx metamodel and 2) to distinguish between dynamic and static libraries since both are declarable with various metamodel objects in ADOxx at the same time and 3) to avoid naming convention errors, that are a result when multiple classes with the same name are defined in one Ecore package. Significantly the latter helped reduce naming transformation steps within the newly created Ecore metamodel. Classes with an identical name but defined in different packages yield no errors on the Ecore metamodel side. Generating a separate ADOxx metamodel package is also necessary for this transformation since many classes in ADOxx inherit from its metamodel classes. The transformation procedure also maps ADOxx metamodel classes to Ecore to guarantee that inheritance relationships behave correctly and don’t throw any errors, e.g., when a certain superclass is missing.

Init Helper Classes This method is supposed to create classes containing helper methods and functions used throughout the application. Those classes are initialized with particular objects (e.g., all packages), and various other classes shall use them. Those helper classes are made publicly available to preserve consistency among the different transformation classes, following the singleton pattern. Whenever another class needs a particular helper, it is referenced by the main transformation class, guaranteeing to offer the single helper that is correctly initialized.

Add classes to packages The first action that this step performs is to create from a source ADOxx class a corresponding Ecore class (`EClass`). The transformation procedure can achieve this by creating a new instance of `EClass` and assigning the class name from the ADOxx class to this new class. After that, the ADOxx class attributes are read to determine whether this class is abstract or not. If the flag is set to "1" in the ADOxx class, the generated class in Ecore is set to abstract by selecting the corresponding Boolean flag’s value to true.

The transformation procedure then adds that particular class to the appropriate package that was created a step earlier.
A static configuration class exists that lists the ADOxx metamodel class names. The name of a possible candidate class is checked upon that list, revealing if it is part of the ADOxx metamodel. If this is true for a certain class, the transformation procedure adds the class to the *ADOxx metamodel package*.

If a class is part of the dynamic library in ADOxx, the transformation procedure should add it to the *Metamodel dynamic package*. ADOxx does not provide this information directly on class-level in the metamodel XML file. The transformation procedure reads the superclass information of a class recursively to obtain this information. Every class in ADOxx inherits at some point from one of the two classes: `__D-construct__`, being the top-layer class of all classes in the dynamic library, and `__S-construct__`, being the top-layer class of all classes in the static library. If any superclass is the abstract class `__D-construct__`, this class has to be part of the Dynamic Library (denoted by the ’__D’ at the beginning of the class name). If the abstract superclass is `__S-construct__`, this class is part of the static library and has to be assigned to the *Metamodel Static package*.

Add supertypes to classes The next step is correctly assigning the superclasses to the created Ecore classes. The procedure firstly iterates through all created classes on the EMF side. For any Ecore class, the corresponding ADOxx class is found in the ADOxx library object (read via JAXB), and then its *super class* attribute is read. This superclass attribute represents the name of the class’s superclass. It is then once again searched for in all created Ecore classes. The Ecore superclass is then assigned to the base class as a superclass.

The superclasses are not assigned a step earlier, right when the classes on Ecore side are created, because some superclasses might not have been initialized at this point. This phenomenon stems from the fact that the items in the ADOxx XML file can have an arbitrary order, not guaranteeing that the procedure reads the actual superclasses into the ADOxx Java model before their base classes. This approach ensures that if a base class has a superclass, it is definitely found and correctly assigned to that base class.

Add attributes to classes In this step, all attributes of an ADOxx class are created and correctly assigned to the corresponding Ecore class. As a first step, the transformation procedure checks if a specific attribute of an ADOxx class is defined in any superclass of the corresponding Ecore class. This is because ADOxx preserves the attribute information for all sub-classes in the metamodel XML file. This check is performed to avoid duplicate creation of an attribute in both super- and subclass, which would yield errors when importing the generated Ecore metamodel to the modeling environment. The attribute creation stops here in positive instances, i.e., the attribute already exists in any superclass. If the check is negative, the ADOxx attribute is then further processed.

Based on the ADOxx attribute type, the transformation procedure creates an `EAttribute`, where its type should semantically match the source type. Most of the ADOxx data types for attributes represent primitive data types in various programming languages. The procedure searches for a suitable equivalent on the Ecore side and, if found, manifests the mapping rule. Different procedures are used for some data types to achieve semantic mapping.

The following listing shows how the different data types are mapped and why we decided to map them in that particular way:

- `string, longstring` → `EString`
  `string` and `longstring` are different classes in ADOxx, the distinguishing feature being a different upper bound of characters for each. The `string` type can hold a maximum string of 3699 symbols, while `longstring` can hold up to 32000 symbols [adof]. The procedure maps both of these types to an `EString`, which represents a string in the programming language Java. `EString` does not have an upper bound, so we decided to map both ADOxx attribute types to this Ecore attribute type.
- `integer` → `EInt`
  Both `integer` and `EInt` represent the primitive datatype integer, so this mapping is semantically suitable.
- `double` → `EDouble`
  The same as for `integer` applies to `double` and `EDouble` respectively.
- `date` → `EDate`
  An `EDate` is in ADOxx represented in the format 'YYYY:MM:DD', which can be represented as well by the Ecore type `EDate`.

- `enum, enumlist` → `EEnum`
  An `enum` in ADOxx can be mapped to an `EEnum` in Ecore, both semantically representing the data structure *enumeration*. However, an enum in ADOxx is simply an attribute that is part of the class where it is defined, just like any other attribute. On the other hand, Enums in Ecore are standalone constructs that have to be created and then assigned as the attribute type on the desired attribute.

  An instance of `EEnum` has to be created and assigned with the correct values to craft an Ecore Enum out of an ADOxx Enum. For each option in an Enum in ADOxx, the transformation procedure creates an `EEnumLiteral`, containing the corresponding literal value. Then the default value, which is set in ADOxx, has to be set for the `EEnum` by selecting the matching `EEnumLiteral` as the default value. Finally, the procedure assigns the corresponding attribute's type with the `EEnum` type that was just created.

  An ADOxx `enumlist` can be created in the same way as just described. The difference between an `enum` and an `enumlist` is the amount of selectable items: While for an `enum` a user can only select one value, for an `enumlist` a user can select a set of available values. This behavior is implemented on the `EEnum` by setting its property `upperBound` to the size of all literals within that enum. In this way, a multi-select `EEnum` can be created, matching semantically the `enumlist` in ADOxx.

  It is worth noting that enum names have to be unique within the same `EPacakge`. To avoid name clashes, every enum is renamed by using the following pattern: *<containedClassName>_<enumName>*.

- `interref` → `EReference`
  Since `interrefs` describe a relationship between two elements, and them following the same definition principle as `EReferences` (i.e. defined within the source class), the best suitable structure in Ecore is not an attribute, but an `EReference`.

  This is performed by finding the source Ecore class based on the ADOxx class's name, finding the target class by the value of `val-c` within that specific `interref` attribute and creating an `EReference`, where the target class that was just extracted is assigned as the reference type.

  Before adding the `EReference` to the source class's `EStructuralFeauture`, the correct cardinalities have to be set. `Interrefs` in ADOxx allow to define a global maximum value on the `REFDOMAIN` level, that applies to all concrete `interref` implementations within that `REFDOMAIN`. However, individual values for a specific `interref` implementation can be set. Their lowest value is chosen, which mimics the behavior on the ADOxx side.

- `default`*(all remaining cases)* → `EString`
  As of now, there are some attributes left that have their attribute values represented as `EString` in Ecore. This has various reasons, some of which

are simply the missing target type in Ecore, but other times, a transformation would require building new data types on the Ecore side, which is not part of this work's scope. However, the goal is to preserve the information, so the user of the target metamodel can use this information to replicate needed aspects themselves. Therefore, we chose the target transformation type of these types to be `EString`.

The transformation procedure does not directly map `datetime` and `time` to temporal types since Ecore only provides the datatype `EDate` by default. This is unsuitable for both types, as only date but no time values can be stored in objects of this type.

`Programmcall` represents routines on ADOxx, which are system-specific and therefore not applicable to Ecore.

Instances of type `table` represent tables in ADOxx, which cannot be easily mapped in Ecore, other than including table tool libraries.

`Expression` contain ADOxx specific expressions to interact with metamodels elements. This type is also not directly applicable to EMF.

`Clobs` are Character Large Objects, frequently used to store arbitrary files like pictures in databases. Since a `clob` is just a concatenation of characters, a transformation to `EString` is very suitable, as the information remains the same, and a model engineer can further process this information in the target platform.

The `AttributeProfileRef` data type, which is used to make references to attribute profiles, is also mapped as an `EString`, since this semantic feature is not present in EMF.

Custom datatypes are mapped to instances of `EString`. `Record Classes` can be used in ADOxx to create structures of (primitive) datatypes, which symbolizes custom datatypes. The same can be achieved with custom datatypes in EMF. A mapping between the two platforms with these two datatypes is therefore feasible. This mapping is not yet realised in the current version of the transformation procedure and will be implemented in a future iteration.

Add relationships to classes The next step is to add the relationships that the ADOxx source metamodel defines to the Ecore metamodel. ADOxx stores the relationship information as `RelationClasses`. Those are located in the XML on the same level as classes. It is then needed to iterate through all `RelationClasses`, read their attributes and values, and create the correct `EReference` on Ecore side.

From the ADOxx `RelationClass`, it is possible to extract the source and target class name with JAXB. The transformation procedure searches for both of these classes on Ecore side. The procedure then assigns the name from the ADOxx `RelationClass` to a new instance of `EReference`, along with the type assigned to the target class. Setting the type of the `EReference` in Ecore indicates, that this type (i.e. this `EClass`) is the target class of this relationship.

The containment property is set to false for every relationship. The containment flag in Ecore determines whether this reference marks a composition, leading to the ability to create new target class instances out of this source class instance on the model level. Since this behavior is not implemented by default in ADOxx (because every element can be added to the canvas directly), the decision fell to set the `containment` flag for all of these relationships to `false`.

The last step is to extract the cardinality information of the corresponding ADOxx `RelationClass`, which is stored in the source ADOxx `Class` and apply the same values to the `upperBound` and `lowerBound` property to the `EReference` object.

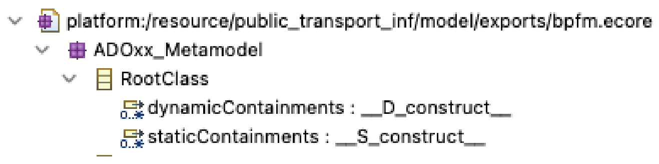
Add basic containments As of now, it would not be possible to create any object on Ecore side for two reasons:

1. There is no root element present
2. There is no composition-relationship present

In order to mitigate these problems, this step creates at first a `RootClass`, which represents the internal root element of any `class` and `relationclass` of ADOxx. Every Ecore model needs a root class to be able to create child instances out of it.

The next step is to add two `EReferences`, one named `dynamicContainments` and the other named `staticContainments`. Both relationships represent a container that possesses the classes of the dynamic library and the classes of the static library, respectively. This is performed by setting the type of the `EReference` to `__D_construct__` and `__S_construct__` respectively. Since every class inherits from either of these classes, the behavior in ADOxx on model creation, where every model instance can be created directly on the canvas, is semantically achieved.

The final two steps are to set the `containment` attribute of these `EReferences` to `true` and add them to the structural features of the `RootClass`. Figure 5.2 shows an excerpt of the achieved behaviour on metamodel level.



```
platform:/resource/public_transport_inf/model/exports/bpfm.ecore
  ADOxx_Metamodel
    RootClass
      dynamicContainments : __D_construct__
      staticContainments : __S_construct__
```

Figure 5.2: Metamodel containment feature of target Ecore metamodel

Resolve duplicate names Since EMF is strongly tied with the programming language Java, and eventually, valid Java code should be generated from the created metamodel, it is necessary to follow the Java naming conventions on Ecore side. This procedure helps resolve issues regarding regular expression for Java classes, packages, and attributes and avoids duplicate names of classes within a package, duplicate attribute names within a class, and duplicate relationships within a class.

This step focuses on resolving duplicate class names and duplicate relationships names. It is important to note that two classes within one package can be considered duplicates by Ecore (yielding a warning) if they have a different amount of underscores in arbitrary order and all other characters and their order remain the same. E.g. the name `SomeClass` is considered as equal with the name `__Some_Class__`. This is especially to be taken care of in this transformation since a lot of ADOxx's classes start with underscores, causing potential problems in this transformation.

The algorithm iterates at this stage through all classes and relationships respectively, reiterates through the same group, and identifies duplicate names. If the procedure finds duplicate names, an increment counter with the value of `2` is defined, which is appended to every identical name and incremented after every use. The complexity of $\mathcal{O}(n^2)$ could be considered too high, given the idea that the transformation procedure could have resolved duplicate names during the creation of the class and relationship. However we chose the iteration approach to avoid maintaining a map throughout the transformation that stores information about the old and the new name of a certain object. We considered that maintaining such maps would result in higher implementation effort and overall complexity.

Output ecore file This step creates an Ecore metamodel file containing all the information created so far, which can be read and interpreted by the Ecore modeling framework. For this purpose, a file called `<source>.ecore` is created, where `<source>` represents the original file prefix of the ADOxx `.xml` metamodel file. Then, the transformation procedure adds the information of these three created packages to the file. This is done via `EcoreResourceFactoryImpl`, which can create a valid Ecore file out of the given Java objects related to Ecore, such as `EPackages`.

5.3 EMF to ADOxx

This section describes the procedure of transforming an Ecore metamodel file and creating an ADOxx metamodel file by introducing to the used technologies, followed by an explanation of the actual procedure.

5.3.1 Used technology

As with the transformation from ADOxx to EMF in section 5.2, the transformation from EMF to ADOxx also uses Java as the transformation program's programming language. Besides the great familiarity with Java, it provides the additional benefit of using tailored libraries for this use case:

The same *Ecore API* [emfc] that is used in the transformation from ADOxx to EMF can also be used here to import the `.ecore` metamodel files into the Java Runtime and access their structure, their object, and their values programmatically. Using this API saves the step of creating an XML to POJO Mapping, as it is described in subsection 5.2.1.

The second external library that the transformation procedure uses is the *ADOxx ALL Public API* [adoc]. This library provides, just as the *Ecore API* [emfc] for an Ecore metamodel does, an API to create, access, and modify objects of an ADOxx metamodel programmatically. This way, it could be combined with the language features of Java, performing the development of the transformation program with minimum effort. Another advantage of this API is that it provides an easy way to create the ADOxx metamodel classes, which are equal to any user-defined ADOxx metamodel, in a few steps, rather than having to create the classes with their attributes individually. Finally, the API can create `.abl` files, which are binary ADOxx metamodel files, that can be imported to an ADOxx environment, thus completing the transformation cycle.

5.3.2 Transformation procedure

The algorithm 5.3 shows the overall procedure of the transformations. The individual methods, which represent the single steps of the transformation, are explained in more detail below.

Set Library Name In contrast to ADOxx, an `.ecore` file can consist of multiple packages, each with a different amount of classes. Furthermore, classes can reference packages in other packages. Splitting metamodel elements into different packages is oftentimes done to add a structure to the metamodel, to separate concerns and to improve readability. However, without providing dedicated meta-information as to which package is the main package (i.e. the package that contains the semantic core features), it is not possible to retrieve this information from arbitrary metamodels. Therefore, the decision fell to assign the file name (omitting the `.ecore` postfix) to the library name, which is performed in this step.

Algorithm 5.3: Overall procedural transformation from EMF to ADOxx

```
    Input: (ecorePackages, file)
1   $setLibraryName(file)$
2   foreach $ePackage \in ecorePackages$ do
3       $addClasses(ePackage)$
4   end
5   $addDependentClasses()$
6   foreach $ePackage \in ecorePackages$ do
7       $addAttributes(ePackage)$
8       $addRelationips(ePackage)$
9   end
10  $processClassesWithMultipleInheritance()$
11  $addMetaInformationToLibrary()$
12  $outputFile()$
```

Add Classes This method is responsible for transforming an `EClass` to an ADOxx `Class` and adding the newly created class to the ADOxx `Library` element. The first step is to find the appropriate superclass of that particular `EClass`, which is a piece of mandatory information that needs to be provided when the transformation procedure creates a `Class`.

While ADOxx does not provide multiple inheritanc, Ecore does. Therefore the procedure then checks, how many direct superclasses (i.e., inheritance level equals one) the concrete `EClass` possesses. The approach of how exactly the procedure performs the multiple inheritance mapping is described in more detail later in this section in the paragraph Process Classes With Multiple Inheritance. However, one crucial step that needs to be performed at this stage is to choose one of the many superclasses and assign it as a superclass to the ADOxx `Class`. The decision fell to choose a greedy approach and take the first item appearing on the list of superclasses to reduce complexity at this stage.

If the Class has more than one direct superclass, the transformation procedure adds it to the list `classesToProcessMultipleInheritance`. Items from this list are handled and resolved at a later stage of this transformation. If an `Eclass` does not have any superclasses, the abstract ADOxx superclass `__D-Construct__` is assigned as the superclass. This property applies to every `EClass` that sits on the highest level of the package hierarchy. Since every class does either possess this property or eventually inherits from a class that owns this property, this step ensures that every class has `__D-Construct__` as its base class. This logic enables the model editor to add this class to the dynamic ADOxx package when creating or editing a model. The dynamic package is used as the default since when interacting with an ADOxx metamodel, users of ADOxx preferably use the dynamic over the static library.

The next step consists of instantiating the class, with the name derived from the `EClass` and the super class derived from the previous steps. Additionally, a procedure is performed, that checks whether the `EClass` has its abstract property set to `true` and assigns the value of `1`, denoting `true`, to the ADOxx `Class`'s *abstract class attribute.*

The transformation procedure then sets a default graph representation string value to finalize the class-creation step. This step is performed because when creating a model element on the ADOxx model editor, the palette would show a blank spot on the model item selection, making it difficult for a user to grab an element from the palette and use it.

Once the procedure fully created and configured a particular class, it adds it to the ADOxx library, which the procedure instantiated in the previous step. This step might sometimes fail, which is described and handled in the next step 'Add dependent classes'.

Add Dependent Classes During the development of the transformation procedure, it is discovered that when trying to iterate through all `EClasses`, the transformation procedure may handle the elements in an arbitrary order. This phenomenon might result from the fact that Ecore reads the defined elements of an `.ecore` file in a random order when using the EMF `ResourceSet`, or that the method `ePackage.getEClassifiers()` does not guarantee the items to be read in the same order they were added. This finding implies that we cannot guarantee that the procedure reads a superclass before its subclass, which results in a Runtime error in the previous step 'Add Classes', when it tries to add a `Class` to the ADOxx library, which does not have its superclass instantiated yet.

To mitigate this problem, all classes that failed in the previous step 'Add Classes' are collected and re-iterated in this step. For every class that is part of this list, the procedure tries again to add it to the ADOxx library, assuming that the superclass was created during the previous process. If this fails again, the conflicting class will be handled in the next iterations of this procedure, and so on, until this procedure can successfully add this class to the ADOxx library. The overall procedure repeats until no more classes are left in this list.

Using this optimistic approach of trying to add a class eagerly, rather than analyzing the capability beforehand, results in the advantage of being minimal in implementation size while also reducing program complexity due to omitting the pre-analysis of hierarchical patterns.

After the transformation procedure finishes this step, every `EClass` has been successfully transformed to an ADOxx `Class` and added to the ADOxx `library`.

Add Attributes When adding attributes to metamodel classes, two things have to be considered:

1. The attributes have to be added to the correct class, deriving name and cardinalities from the source class
2. The Attribute Graph representation string has to be edited in the corresponding ADOxx class to enable editing of attributes on model level

For handling the first issue, the procedure iterates through all available attributes of a particular `EClass`. Then the target type is derived from the source type. We chose the following mapping to replicate the different elements from source to target platform semantically:

- `EString` → `longstring`
  An `EString` from Ecore is mapped to a `longstring` in ADOxx. While ADOxx supports both the data types `string` and `longstring` (both differing in their maximum character size, the latter being higher), we decided always to choose a `longstring` for this transformation. This way, it is more likely to avoid errors of reaching the maximum character cap. ADOxx limits the `longstring` datatype to 32000 characters, so it is still possible, although improbable, to reach this limit since Ecore does not have an upper size for literal values represented as `EString`. A procedure to handle this issue, which is not the scope of this work, would be to create multiple attributes with an incrementing index and partition the literal value of those attributes.
- `EInt` → `integer`
  Since both `EInt` and `integer` represent the primitive datatype integer, choosing this mapping is the most fitting decision for this transformation step.
- `EDouble` → `double`
  Since `EDouble` and `Double` represent the primitive datatype double, this mapping is also trivial.
- `EDate` → `date`
  An `EDate` can be easily converted to a `date` data type in ADOxx since both represent the temporal value date.
- `EEnum` → `enumeration, enumerationlist`
  When it comes to Enums, it has to be decided if the appropriate target type is an `enumeration` or an `enumerationlist`. The procedure achieves this by checking the attribute *upperBound* of a particular `EAttribute`. If this upper bound is greater than one, the target type is an `enumerationlist`. If it is exactly one, it is an `enumeration`. Ecore allows the metamodel engineer to define how many different literal values are choosable (e.g., providing an `EEnum` with four different literals, where at max two can be chosen). This behavior cannot be replicated in ADOxx, since an `enum` allows the user to choose exactly one literal, and an `enumerationlist` allows to select at max as many literals as this object contains. However, mapping `EEnum` with an upper bound higher than one to an `enumerationlist` is semantically the most fitting transformation procedure since the semantic attribution of multi-selection of literals can be preserved.
- `default` *(all remaining cases)* → `longstring`
  We considered mapping all remaining attribute types to the type `longstring`. Ecore supports the creation and the use of arbitrary types to assign them

to an `EAttribute`. In ADOxx, this mapping could be realised with the help of `RecordClasses`. Both custom datatypes in Ecore and record classes in ADOxx allow to define multiple constructs of simple other (primitive) datatypes. This step is not yet part of the transformation procedure an will be implemented in a future iteration.

We decided to preserve the information of a *default* attribute as a `longstring` to conserve the semantic meaning of the source type in the target environment and to help the user take appropriate actions when handling this data type on ADOxx side.

After selecting the type, the default value of the literal is derived. If the `EAttribute` has a default value defined, it is also used for the newly created `Attribute` in ADOxx. If it is not defined, default literals are chosen based on the literal type. This is for string values *„Default value“*, for numeric values the number *1* or *1.0* for integers and floating point types respectively, for temporal values the *1st of January 2022* in its appropriate representation, and for enumerations the *first value* present in the list of enumeration literals.

The ADOxx `attribute` is then created with the information of name, identifier, and default value derived in the previous steps and assigned to the containing ADOxx `Class`. Additionally, the flag *isClassAttribute* is set to `false`, and the default facets are set via the ADOxx_ALL API [adoc].

For enums, the transformation procedure applies a special routine: It initially creates an attribute without a default value. Literals in ADOxx have to be concatenated as a string, where every literal is separated with an '@' symbol. After creating this string from the derived Ecore literals, it is then added to the special facet `EnumerationDomain`, thus making it a valid ADOxx `enumeration`. A second attribute must be created, with the same name, only with the default value set, to add the default value to this enum. This combination of duplicate attributes is necessary to register the default value to the enum and avoid warnings when trying to import the metamodel file in ADOxx. Code Example 5.2 shows such an example construct, where the enum *Type-Selection* is defined along with default value literal being *type-3*.

Finally, the attribute representation of the class has to be solved. The transformation procedure handles this in parallel to the routine of creating the ADOxx `attribute`. The creation of the attribute representation is needed in order to enable the user to edit the different attributes in the model editor. While the transformation procedure transformed an attribute, it crafted a special string containing the `NOTEBOOK` keyword along with a chapter and description information, followed by the attributes represented as `ATTR`, adding the attribute name at the end. Every attribute information is also added to the string in this procedure to mimic the behavior of Ecore to ADOxx, where every attribute is editable when creating a certain model object. After handling all attributes for a class, the transformation

```
ATTRIBUTE <Type-Selection>
TYPE ENUMERATION

   FACET <EnumerationDomain>
   VALUE "type-1@type-2@type-3"

   FACET <MultiLineString>
   VALUE 0

   FACET <AttributeHelpText>
   VALUE ""

   FACET <AttributeRegularExpression>
   VALUE ""

ATTRIBUTE <Type-Selection>
VALUE "type-3"
```

Code Example 5.2: ADOxx ALL file excerpt showcasing enum creation

```
   NOTEBOOK
   CHAPTER "Description"
   ATTR "axlesCount"
   ATTR "condition"
   ATTR "tankSizeInLiters"
   ATTR "vehicleName"
```

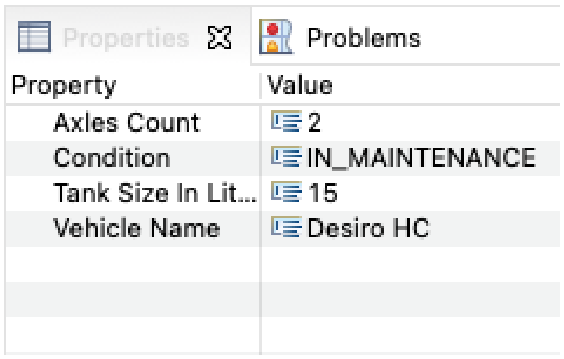
Code Example 5.3: Sample ADOxx attribute representation value

procedure adds the crafted string to the *AttributeRepresentation* attribute of that particular ADOxx `Class`.

A sample attribute representation is referenced in Code Example 5.3, where the attributes *author*, *age*, and *location* are added to the representation. Figure 5.3 shows additionally a corresponding comparison of the attribute modification interface for a source EMF model instance and a target ADOxx model instance.

Add Relationships The next step consists of adding the relationship information from the source Ecore metamodel to the target ADOxx metamodel. For this, the `EReferences` of a particular `EClass` are derived and then iterated. For every relationship, the procedure first checks whether it is containment or not (by checking the *containment* flag of the `EReference`). If it is a containment, the relationship is specially treated with the `Composition Converter`, which is further described in the upcoming item Process Compositions.

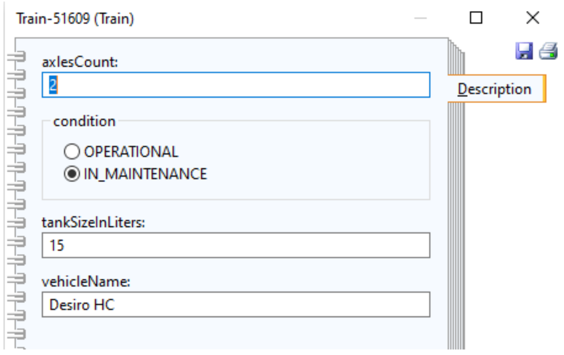
If it is a regular relationship, however, the information of its name and target



Properties Problems

Property	Value
Axles Count	2
Condition	IN_MAINTENANCE
Tank Size In Lit...	15
Vehicle Name	Desiro HC

(a) Attribute editor in EMF



Train-51609 (Train)

axlesCount: 2

condition
○ OPERATIONAL
● IN_MAINTENANCE

tankSizeInLiters: 15

vehicleName: Desiro HC

(b) Attribute editor in ADOxx

Figure 5.3: Comparison of attribute editing interface of both metamodeling platforms

class is derived from the `EReference`. Since in Ecore, a reference is contained within the `EClass` where it's defined, the transformation procedure can trivially derive the source class name of the to-be-created ADOxx `Relation Class` from the currently inspected `EClass`. After gathering this information, the ADOxx `Relation Class` is then created and added to the scope of the dynamic ADOxx library.

The last step is to process the cardinalities of an `EReference`. The transformation procedure defines the cardinality information in Ecore within the `EReference`, where it derived its information from. In ADOxx, the cardinalities must be defined in either of the two involved classes (i.e., the source or the target class). ADOxx allows to set specific attributes for cardinalities, namely *min-outgoing*, *max-outgoing*, *min-incoming* and *max-incoming*. The first two are suited to define semantic constraints on the source class, while the two latter can be used to define constraints on the target class. We decided to only define the cardinalities *min-outgoing*, *max-outgoing* within the source class, by deriving the information from the *lowerBound* and *upperBound* information of the corresponding `EReference`. We chose the source class to hold the constraints information, as it is the most appropriate semantic place. It's easier for a user to understand how many instances of a relationship to another class are allowed rather than how many incoming relationships to a target class are permitted. Additionally, we can semantically replicate the behavior of Ecore, where the cardinality information is derived from the source and not the target `EClass`, since Ecore defines the `EReference` within that particular source `EClass`.

A final consideration is to make the cardinalities consistent within source and target ADOxx `Class`, by setting the *min-outgoing* value of the source class as the *min-incoming* value of the target class, as well as the *max-outgoing* value of the source class to the *max-incoming* value of the target class. However, this approach is redundant since providing the information on the source class alone is sufficient to produce the correct semantic behavior. Additionally, when considering later refactoring of the generated metamodel, it is easier to define cardinalities only within one ADOxx `Class`, which would present a single point in the metamodel where cardinality information has to be changed, rather than having to worry about multiple occurrences in the metamodel that have to be kept consistent. Since relation class names have to be unique, the relationships are renamed, following this specific pattern: *sourceClassName-ecoreReferenceName-targetClassName*.

Process Compositions If a relationship is determined to be a composition, the transformation procedure handles it in this step of the transformation procedure. The first step is to determine all the instantiable classes of a particular compositum, i.e., all composites. The compositum information is derived from the `EType` information of the composition's `EReference`, which represents the source class of the relationship.

For each composite instance, the compositum `interref` information has to be updated. Every composite needs to have an `interref` relationship between its compositum and itself to be properly handled through the composition procedure written in AdoScript. Therefore, the transformation procedure searches for the *composedInstances* attribute in the compositum class. This attribute has to be a well-formed string that contains all the information of the various `interrefs`. If no prior definition exists, the procedure creates this attribute, simultaneously assigning an upper bound by setting the '`max:`' string value to the value of *upperBound* property of this attribute defined on Ecore side. If the `interref` attribute already exists, the procedure appends the information of the new `interref` to the already existing attribute facet. If the facet needs editing, a special line break escape procedure is executed, which replaces line breaks that are hard-coded as `\n` by the ADOxx ALL API [adoc]. Ignoring this step might lead to the value of the *composedInstances* attribute getting malformed, resulting in an invalid `.abl` file, which the user cannot import to ADOxx.

An example *InterRefDomain* value can be seen in Code Example 5.4, where numerous composite-relations are defined for a certain class. The last two steps consist of editing the *compositeClasses* and *compositumClass* strings, that are added as attributes to the class `__LibraryMetaData__`. This is done by appending the corresponding class names to a string and separating them with a comma. This information is needed to handle the creation and deletion of composition elements properly.

Process Classes With Multiple Inheritance This step handles the correct assigning of attributes for multiple-inheritance classes, as well as creating the appropriate structures for dealing with incoming relationships of a multi-inheritance class.

Figure 5.4 gives an overview of an example multiple-inheritance pattern and how it is translated during the transformation. In the previous step, 'Add relationships',

```
REFDOMAIN
OBJREF
    mt:"BibTeX"
    c:"Misc"
    min:0
OBJREF
    mt:"BibTeX"
    c:"Unpublished"
    min:0
OBJREF
    mt:"BibTeX"
    c:"Proceedings"
    min:0
OBJREF
    mt:"BibTeX"
    c:"InProceedings"
    min:0
OBJREF
    mt:"BibTeX"
    c:"MasterThesis"
    min:0
OBJREF
    mt:"BibTeX"
    c:"PhDThesis"
    min:0
```

Code Example 5.4: Sample Attribute InterRefDomain value of a compositum

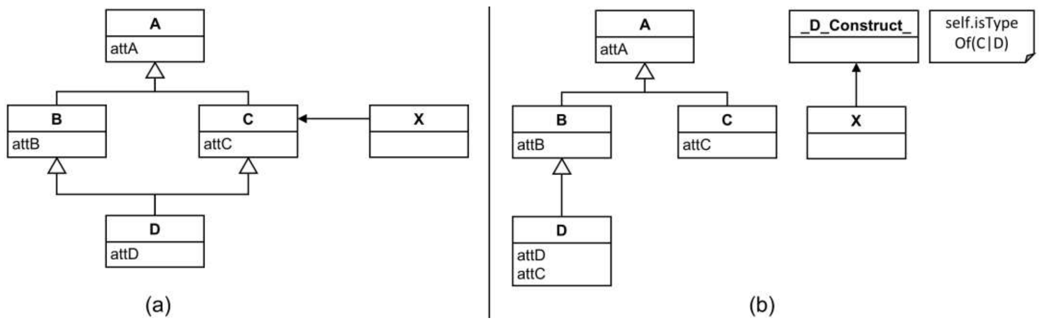


Figure 5.4: Multi-inheritance example (a) and the adapted expansion pattern (b) [BWA21]

classes that have more than one superclass on Ecore side were implemented as follows: Only one of the available superclasses was directly assigned as a superclass on ADOxx side. This stems from the fact that ADOxx only supports single but not multiple inheritance. The attributes of the other classes that are not used as superclasses must be added to that particular ADOxx class. Therefore, the attributes of every other superclass are determined and put in a list. For each found element in that list, the attribute is added to the appropriate ADOxx `Class`:

First, the transformation procedure ensures that there is no attribute with the same name and the same type already defined in one of the superclasses of the ADOxx `Class`. If this step would be ignored, the resulting `.abl` file would be invalid, resulting in a non-working import in ADOxx. To determine whether that attribute is already used in one of the superclasses, the procedure iterates through all superclasses and retrieves all attributes. If any of these attributes is semantically equal to the attribute desired to be added, the procedure ignores it and continues with the next attribute.

Secondly, the procedure has to ensure that no attribute in a subclass is equal to the attribute that has to be added. This scenario might occur in certain metamodels because the procedure might read classes and attributes in an arbitrary order, not corresponding to the actual hierarchy pattern that the Ecore metamodel provides, as already described in the section Add dependent classes. If the transformation procedure finds a semantically equal attribute in one of the sub classes, it is removed from there and added to the currently handled class. This step ensures as well that no malformed `.abl` is generated in this process.

The next step is to handle the incoming references of superclasses of a particular Ecore class, that are part of a multi-inheritance pattern. The transformation procedure has to handle incoming references specially, i.e., assigning them to the abstract superclass `__D_construct__` and limiting the classes that can be part of the relationship. This procedure has to be performed to achieve semantic equality of the behavior of incoming relation classes of a multi-inherited class:

First, all the incoming relationships of a particular superclass are retrieved. The transformation procedure does this by checking if any `EReference` has as its target class this particular class assigned. For every relationship, a new ADOxx `Relation Class` is created, which has the `EReference`'s source class as its source class, and the abstract class `__D-construct__` as its target class. The procedure then adds the created relationship to the ADOxx library. The next step consists of modifying the attribute string *multiInheritanceClasses*, which becomes part of the abstract class `__LibraryMetaData__`, which is needed to handle the multi-inheritance relationship functionality on ADOxx side properly. The last step consists of adding or modifying the attribute *validIncomingRelationClasses* with the information of the classes that are part of the incoming relationship (in Figure 5.4 this would be class C and class D). The procedure achieves this by adding the name of every valid incoming `Relation Class` to this attribute, separated by a comma.

There are alternative inheritance patterns available, as described in the work of [CMR02], which represent an alternative solution to resolve the problem of transforming a multi-inheritance to a single inheritance pattern. However, we decided to choose this overall procedure, as it is possible to preserve all the static information present in the source metamodel (i.e., the attributes and relationships) but also to create semantic equivalence in the target platform ADOxx, i.e., the same behavior as in the source metamodel, as we could use AdoScripts to introduce constraints and rules to the target metamodel.

Add Meta Information to Library In this step, the transformation procedure adds additional information to the metamodel to

1. register the metamodel elements to the metamodel domain and
2. enable the correct behavior of special transformation features such as composition and multi-inheritance

The first step of this procedure is to edit the `MODELTYPE` of the library and add it to the `MODI` library attribute. Finishing this procedure enables the possibility to create new instances of model elements based on their metamodel definitions in the model editor. Since only instantiable classes are addable to the `MODELTYPE` attribute, the procedure iterates through every class that is present in the ADOxx library, confirms that the class is non-abstract, and adds a new line to the `MODELTYPE` string with the scheme `INCL` followed by the class name. An example `MODELTYPE` value for the library *BibTeX*, corresponding to a distinct Ecore metamodel, is given in Code Example 5.5.

```
MODELTYPE "BibTeX"
INCL "BibTeXFile"
INCL "Author"
INCL "Unpublished"
INCL "Proceedings"
INCL "InProceedings"
INCL "Misc"
INCL "PhDThesis"
INCL "MasterThesis"
```

Code Example 5.5: Sample MODELTYPE value of an ADOxx library

The transformation procedure then has to add the names of the `Relation Classes` defined in the ADOxx metamodel in the same fashion, which it performs in the following step.

The next step is to edit the `ExternalCoupling` information of the ADOxx library to enable the correct behavior of compositions and multi-inheritance. In this attribute, the procedure concatenates the previously defined code for handling compositions with the special events corresponding to those files. This is done by firstly defining the event, e.g., ON_EVENT „AfterCreateModelingNode“, and secondly enclosing the code that the procedure read from the stored code file packaged with the application between two braces.

For this transformation, the following events are used:

- **AfterCreateModelingNode**
  The code inside this event is triggered when a model user creates a new modeling node on the model canvas. The code responsible for checking and verifying the presence of an existing compositum class for a given composite class in the model instance is defined here.
- **BeforeDeleteInstance**
  This event is triggered when a model user deletes a certain element from the model editor. Here, the code is embedded that checks whether the class that is about to be deleted is a compositum class, then searches for all-composite classes, and deletes them from the model.
  The pseudo-code algorithms in algorithm 5.4 illustrate these two composition procedures on the model level in ADOxx.

Algorithm 5.4: AdoScript code for handling composition [BWA21]

```
Input: classid, objid, and modelid of the object o to be created
ON_EVENT "AfterCreateModelingNode"
compositeClasses ← LibraryMetaData.compositeClasses()
if compositeClasses contains classid then
    compositumClass ← o.compositumClass()
    availableCompositumObjects ← GET_ALL_OBJS_OF_CLASSNAME(modelid,
      compositumClass)
    if availableCompositumObjects.size() > 0 then
        selectedCompositumObject ← LISTBOX(availableCompositumObjects).selection()
        ADD_INTERREF(selectedCompositumObject, o)
    else
        DELETE_OBJ(o)

Input: classid, objid, and modelid of the object o to be deleted
ON_EVENT "BeforeDeleteInstance"
compositeClasses ← LibraryMetaData.compositeClasses()
compositumClasses ← LibraryMetaData.compositumClasses()
if compositumClasses contains classid then
    for Composite c : o.composedInstances() do
        DELETE_OBJ(c)
    end
else if compositeClasses contains classid then
    compositumClass ← o.compositumClass()
    availableCompositumObjects ← GET_ALL_OBJS_OF_CLASSNAME(modelid,
      compositumClass)
    for Compositum com : availableCompositumObjects do
        if com.composedInstances().contains o then
            REMOVE_INTERREF(com, o)
    end
```

- **AfterCreateModelingConnector**
  This event is triggered when a user adds a new relationship to the model, i.e., two classes are connected by a `Relation class`. The code embedded in this event is responsible for checking that the relationships derived from a multi-inheritance structure in Ecore behave correctly, i.e., are allowed between the desired two classes.

The algorithm 5.5 gives a pseudo-code overview of the overall procedure that models based on the transformed metamodel perform.

Algorithm 5.5: AdoScript code for handling multiple inheritance [BWA21]

```
Input: classid, relationid, modelid, toObj, and fromObj of the relation r to be created
ON_EVENT "AfterCreateModelingConnector"
multiInheritanceClasses ← LibraryMetaData.multiInheritanceClasses()
if multiInheritanceClasses contains classid then
    validIncomingRelationClasses ← toObj.validIncomingRelationClasses()
    if !validIncomingRelationClasses contains classid then
        DELETE_CONNECTOR(r)
    end
end
```

The last step is to add the names of the classes needed for the scripts in the previous step to work to the metamodel. We decided to put this information as new attributes into the abstract class `__LibraryMetaData__` since it is semantically most suitable to define this information as metadata that is needed for the library to perform correctly. Three attributes where created, namely *compositeClasses*, *compositumClasses* and *multiInheritanceClasses*, all containing the strings that were acquired in the previous steps of the transformation process and that represent the desired classes respectively.

Output File This last step converts the generated metamodel defined in Java through the ADOxx_ALL API [adoc] to `.all` and `.abl` files. The `.all` file is a human-readable format that represents the created metamodel. While this format is easy to read and understand, ADOxx does not offer an option to import it to its metamodel editor.

The transformation procedure must create an `.abl` file to allow users to import generated metamodels into ADOxx. The transformation procedure generates this file by using the *ADOxx ALL to ABL web service* [adod], which takes an `.all` file as input and converts it into an `.abl` file. The code for this transformation is closed-source, meaning that a local execution without using the web service is not possible. The API conveniently invokes this web service, which avoids the definition of writing additional code to perform this web-services call.

5.4 Scenario

This chapter is accompanied by an illustrative scenario based on an example of a Fleet Management system to gain a better understanding of the transformation and show that a metamodel engineer can use the transformation procedure for an actual use case. The term "illustrative scenario" is adapted by the authors in [PRTV12], that introduced it in the context of evaluation methods. According to the authors, "[illustrative scenarios] apply the artifact in a synthetic or real world situation to demonstrate its utility" [PRTV12], which resembles the procedure that is taking part here.

This illustrative scenario includes all the features that both metamodel platforms share (e.g., core modeling concepts) and emphasize the differences between the two (e.g., different inheritance patterns). The illustrative scenario covers those differences, especially to reveal that the transformation procedure can find a solution for mapping differences in core features and resolving edge cases.

The UML metamodel in Figure 5.5 functions as a blueprint to create the metamodel on a specific platform. We used UML for this purpose since its instances are easily readable, generic (i.e., metamodels on either platform can be easily created out of it due to common first-class concepts), and familiar with the general audience of model engineers.

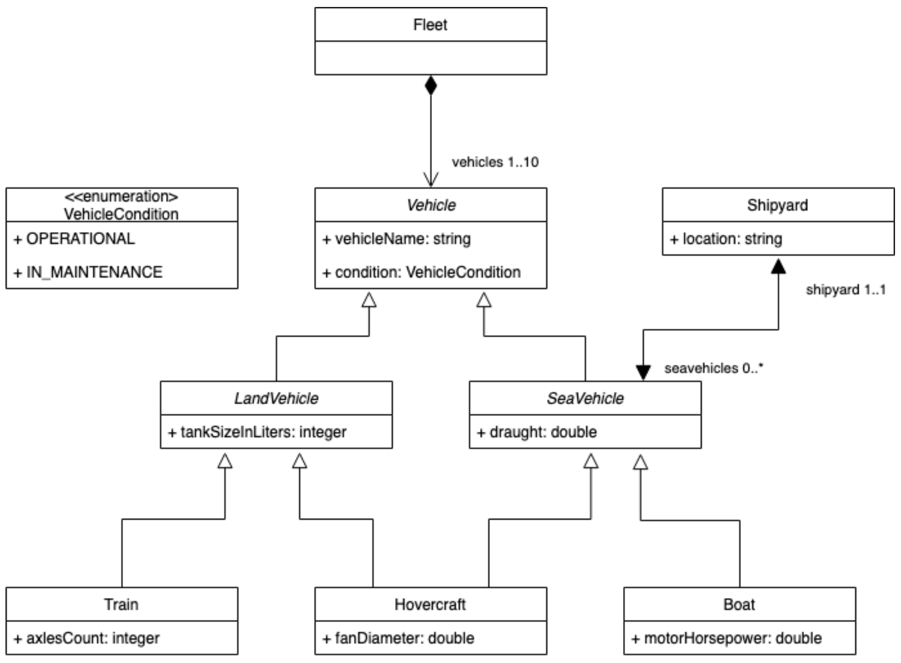


Figure 5.5: UML class diagram metamodel of the illustrative scenario

The illustrative scenario consists of a `Fleet` class, which holds all information about the different Vehicles of a fleet. It has a composition-relationship to the class `Vehicle`, which means that instances of `Vehicle` can only be created when an instance of `Fleet` exists, as well as instantiation bounds of at least one and at max ten. The `Vehicle` class is abstract and holds two variables: One is the *vehicleName*, a string to represent the name, the other is *condition*, which is an Enum, where the values can be either *OPERATIONAL* or *IN_MAINTENANCE*. Two abstract classes inherit from `Vehicle`, namely `SeaVehicle`, which holds a numeric floating point attribute of *draught* and `LandVehicle`, which holds a numeric integer attribute of *tankSizeInLiters*.

With this constellation, the core modeling features of class and attribute are already covered with all attribute types (including Enums) and single inheritance. The instantiable class `Shipyard` is added to the metamodel, and two unidirectional relationships between

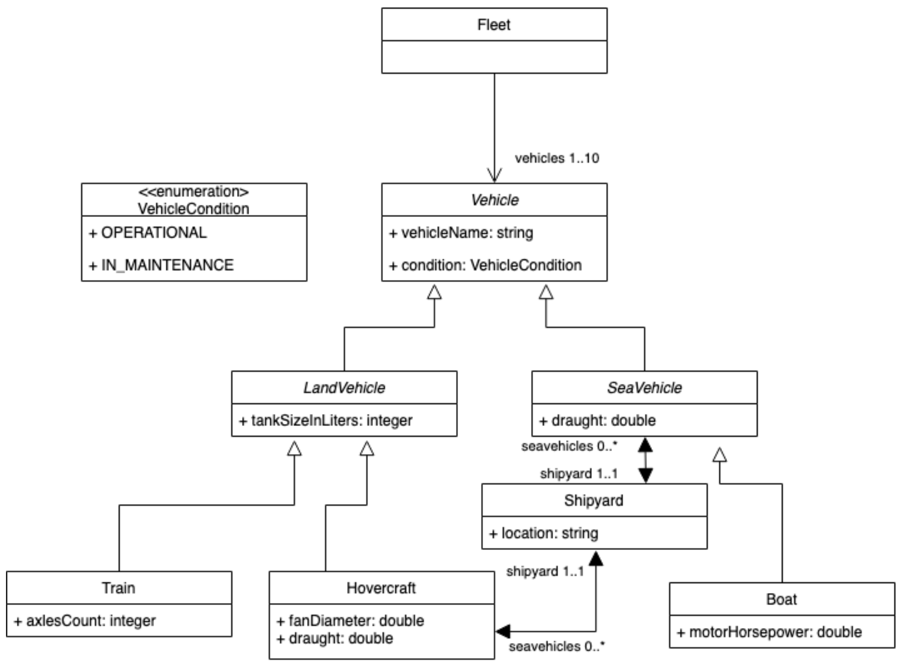


Figure 5.6: UML class diagram metamodel for the direction ADOxx to EMF

`Shipyard` and `SeaVehicle`, both with different cardinalities, to cover the core modeling feature of relationships and cardinalities.

The `Boat` class is then added, holding the attribute *motorHorsepower* and inheriting from `SeaVehicle`. The `Train` class is added as well, holding the attribute of *axlesCount* and inheriting from `LandVehicle`. These two classes are now instantiable, inheriting from different and common superclasses, which helps to reveal the semantic correctness of the transformation when it comes to single inheritance pattern and the related attribute assignment.

Finally, the `Hovercraft` class is added, also instantiable, with a parameter *fanDiameter*. This class is special in that it inherits from both `SeaVehicle` and `LandVehicle`, trying to enforce the correct multi-inheritance behavior of the transformation project.

5.4.1 ADOxx to EMF

Since ADOxx does not support multiple inheritance, the illustrative scenario that we introduced in the previous section in Figure 5.5 is adapted slightly to preserve the semantic correctness of the proposed metamodel while also creating a metamodel that conforms to the rules of ADOxx.

In Figure 5.6, it can be seen that the inheritance of the class `Hovercraft` to `SeaVehicle` has been deleted. The attribute *draught* is derived from the class `SeaVehicle` and directly added to the class `Hovercraft`. A new bidirectional relationship between

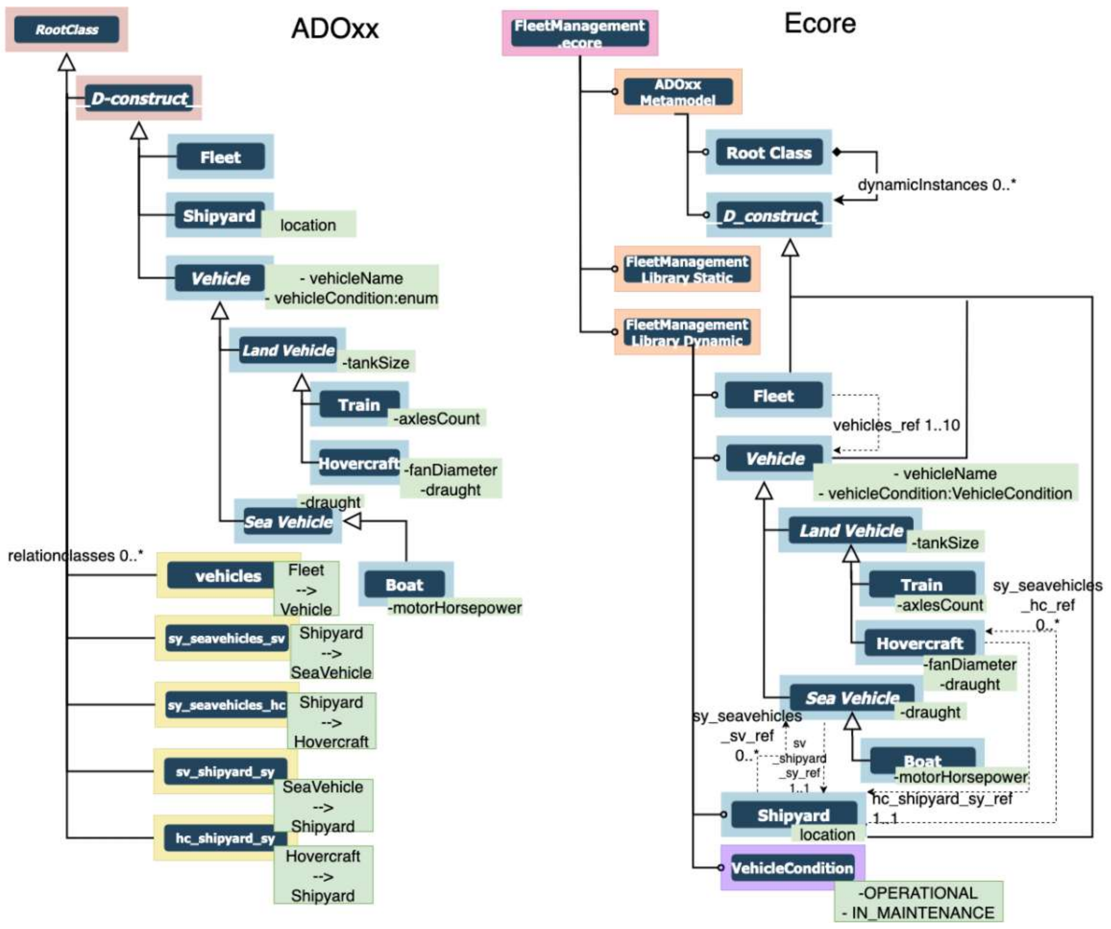


Figure 5.7: Transformation scenario: From an ADOxx to an Ecore metamodel

Hovercraft and Shipyard is added, based on the existing relationship between SeaVehicle and Shipyard, adapting its cardinalities to preserve these relationships which the metamodel would have otherwise lost. Finally, since ADOxx does not support composition directly, the composition between Fleet and Vehicle is transformed to regular relationships, while also preserving the cardinalities.

The transformation in Figure 5.7 shows an excerpt of the transformed metamodel. Lines with a circle at the end represent a "target element is part of source element" relationship, dashed lines with an arrow on either end represent a regular relationship, arrows with a diamond on one end represent a composition, and the inheritance arrows represent a "target element is a subclass of source element" relationship.

Inside the Ecore metamodel, three Packages (in orange) have been created, namely ADOxx Metamodel, Fleet Management Library Static and the library Fleet Management Library Dynamic.

The package ADOxx Metamodel contains all classes that are part of the ADOxx metamodel, including `__D_construct__`, as well as the implicit superclass of every element in ADOxx, the `RootClass`. This class contains a composition relationship with the class `__D_construct__` to be able to create an arbitrary number of elements underneath this level. For one, this copies the behavior of ADOxx, where a model user can add every element on a single plane (the canvas). For the other, it enables the Ecore user to create model-element instances. Only if an element is part of a composition can it be created through the model editor.

The package `Fleet Management Library Static` contains only classes of the static library. This package exists to avoid name clashes. Splitting classes with the same name but in different packages yields no error for an Ecore metamodel.

The `Fleet Management Library Dynamic` package contains all transformed classes. *Italic* classes are abstract classes, while all other classes are instantiable. We preserved the inheritance pattern that we initially provided during the transformation. Also, the inheritance pattern involving the class `__D_construct__`, that is present in ADOxx, is preserved here. This step is important in order to perform the correct instantiation of the other metamodel classes.

The attributes are also added and mapped with the appropriate name and type inside the corresponding class from the source metamodel. Additionally, the transformation procedure creates an enum that is part of `Vehicle` on ADOxx inside the dynamic package. The literals of type `EEnumLiteral` have been adapted and added to this object. This enum is then referenced in the attribute *vehicleCondition* of the class `Vehicle` on Ecore side.

Finally, for every `RelationClass` in ADOxx, relationships inside the corresponding source class have been created. The procedure preserved the cardinalities, though it adapted the relationship names with the string-appendix `_ref` to mitigate name clashes with existing attributes.

5.4.2 EMF to ADOxx

The illustrative scenario for this direction is adapted from Figure 5.5 and needed no special transformation as in the previous subsection 5.4.1. As shown in Figure 5.8, the transformation takes an Ecore Metamodel and transforms it to an ADOxx metamodel.

The orange element on Ecore side, the `Fleet Management` is the package of this Ecore metamodel. The blue elements on either side represent classes. Abstract classes have an *italic* label, while instantiable classes have a normal label. On either side, attributes of a class are represented through green elements. The violet element on Ecore side represents an Enumeration object. The red elements on ADOxx side are metamodel elements of the ADOxx Library, which are pre-generated and non-deletable. Yellow elements on ADOxx side represent `Relation Classes`.

The diamond arrow on Ecore side represents a composition. The dashed arrow on Ecore side between two classes represents an `EReference`, i.e. a relationship. Dashed arrows

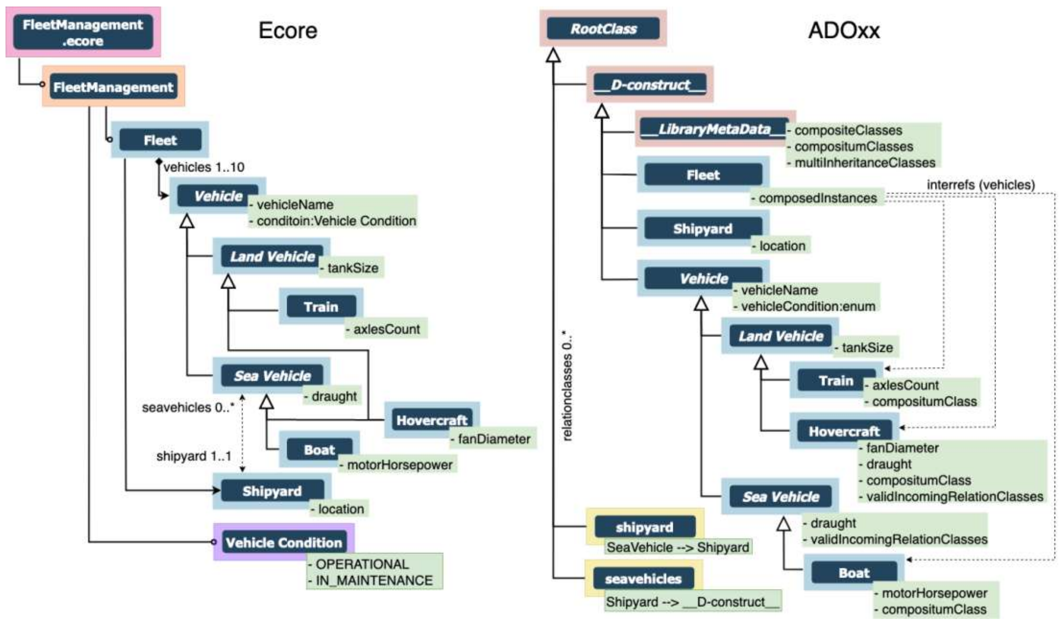


Figure 5.8: Transformation scenario: From an Ecore to an ADOxx metamodel

on ADOxx side also represent a relationship, but of type `Interref`. The inheritance arrows between two classes resemble a subclass relationship.

Out of the ecore package element `FleetManagement`, the ADOxx library is created and named. All classes from Ecore are created on ADOxx, deriving the name and their instantiation-capability from the Ecore class. The transformation procedure mapped all attributes by creating and adding them to the corresponding ADOxx class based on the definition on Ecore side. For enums, a special procedure applies, which this work describes in a subsequent section.

Since the `Fleet` object is a compositum element, the transformation procedure created a new attribute ADOxx side, namely the *composedInstances* attribute, that holds the information of all composite elements of this compositum class. For every composite class, the procedure created a new attribute, namely *compositumClass*, that holds information about every compositum class it is part of.

The class *Vehicle* contains an internal attribute reference to the `EEnum: Vehicle Condtion`. On ADOxx side, this enum is represented in the class `Vehicle`, where simply an attribute of type `enum` is added. In Ecore, the class `Hovercraft` inherits from both the class `LandVehicle` and `SeaVehicle`. On ADOxx side, the class `Hovercraft` only inherits from one of these classes, namely `LandVehicle`. The attribute that is part of `SeaVehicle` on Ecore side is duplicated and added to the class `Hovercraft`.

The bidirectional relationship between the classes `SeaVehicle` and `Shipyard` can be viewed as two unidirectional relationships. It follows that 2 `RelationClasses` have to be created on ADOxx side, one named `shipyard`, which represent the relationship between `SeaVehicle` and `Shipyard` and one named `seavehicles`, which represents the relationship between `Shipyard` and `SeaVehicle`. Since the latter represent a relationship, where the target class is a multi-inheritance class that is not directly referenced as super class in the ADOxx metamodel (i.e. `SeaVehicle` is not the super class of `Hovercraft` on the ADOxx side), the `RelationClass` needs to have the target class as `__D-construct__`, in order to enable the valid handling of incoming relation classes of multi-inherited classes. The cardinalities on Ecore side are applied on the source classes on ADOxx side. On the classes `Hovercraft` and `SeaVehicle`, the attribute *validIncomingRelationClasses* is added to fully enable the handling of incoming relation classes based on multiple inheritance.

CHAPTER 6

Evaluation

In this evaluation chapter, the results of an evaluation-based experiment for a sample of representative size are collected and analyzed. The procedure of evaluating the transformation consists of taking every source metamodel of a framework, performing the transformation procedure and collecting the results of the metamodels before and after.

The evaluation is split into a syntactic part in section 6.1 and a semantic part in section 6.2. The syntactic evaluation analyzes the different structural features of metamodels before and after the transformation. Those findings reveal how the transformation procedures map elements like classes, attributes, and relationships to the other framework. Through this step, the possibility arises to recognize and evaluate certain patterns (e.g., a larger size of relationships in the target platform) and conclude different transformation findings. The syntactic evaluation also describes how it is possible to make statements about the validity of the transformation, i.e., by knowing how the transformation works internally, it is possible to validate the size of the elements created to make sure that the transformation behaved correctly.

The semantic evaluation helps to find out, whether the source and the target metamodels behave semantically correct. This means that this step consists of manually checking a subset of source and target metamodels and seeing if the semantic features like names, composition and inheritance patterns were mapped semantically equal from one platform to another.

We chose the samples from a variety of sources: For ADOxx, we took a set of 45 metamodels from the analysis of [Bor18]. In this work, the author conducted an evaluation-based analysis of those metamodels. A set of 29 metamodels are chosen for the EMF evaluation from the ATL-Zoo for EMF at [emfb]. We chose only metamodels that are not dependent on other metamodels (.ecore files), which would imply an additional amount of logic to be implemented in the evaluation program. From the given super set of more than 100 Ecore metamodels, only 32 (excluding a custom made metamodel not part of the super set) could be used for this experiment.

In both cases, metamodels from different domains and a great variety of characteristics are chosen to show the feasibility of the transformation on different domains and metamodel sizes. Those characteristics include the size of classes, relationships, inheritance depth, and containments. This broad scope of metamodels can yield interpretable results regarding the impact of metamodel size on syntactic structures and semantic correctness. Table 6.1 gives an overview of the source metamodels that the evaluation section used per direction.

Table 6.1: Metrics overview of the source metamodels used in the experiments

	ADOxx			**EMF**		
	Min	**Med**	**Max**	**Min**	**Med**	**Max**
Classes	5	30	180	1	7	93
Abstract Classes	0	2	24	0	1	12
Relations	1	11	81	0	2	59
Compositions	0	2	13	0	2	36
Attributes	2	86	1165	1	6	64
Inheritance Depth	1	3	6	0	1	4
Multi Inheritance Classes[1]	-	-	-	0	0	4
Enumerations	0	17	270	0	0	7

[1] Multi inheritance is not supported in ADOxx

The set of ADOxx metamodels is generally more expressive since it offers a greater variety of metamodels with different characteristics. Most attributes are usually more abundantly present in metamodels of the ADOxx sample, as denoted by the median values of those characteristics. One could conclude that the Ecore source metamodels are much smaller than their ADOxx counterpart in terms of the size of internal structures. But looking at the maximum values for the same characteristics yields that within the set of Ecore metamodels, there are still metamodels with a similar amount of characteristics as in the ADOxx sample available.

The characteristic *Classes*, *Relationships*, *Attributes* and *Enumerations* gives an overview of the amount of first-class concepts present in either sample. A variety of these characteristics gives a general picture of the metamodel size and the amount of different items available. The amount of *Abstract classes* helps to verify the correct mapping of the abstract features of a class in either metamodel environment.

The *Inheritance Levels* on either platform show how well the transformation behaves with inheritance patterns. Especially for the EMF samples, it is essential to include metamodels with multiple relationships (noted by the characteristic *Multi Inheritance* *classes*) as well as incoming relationships, which would yield important insights about the correct behavior of transforming multiple inheritance structures from EMF to ADOxx. While most metamodels do not have any multi-inheritance class, few metamodels have a significant amount, with the highest present amount of classes being four.

But also, including metamodels from EMF that contain many compositions is important for this evaluation. This way, it is possible to verify that the transformation procedure manages to represent composition features syntactically and semantically correct in ADOxx.

6.1 Syntactic Evaluation

The syntactic evaluation of the transformation helps to gain insight into the nature of the generated artifacts. As part of the procedure, we chose different categories based on previous work conducted in metric-based evaluation on metamodeling platforms: In [KHK11], the authors compared the various metamodeling features of the platforms ARIS, Ecore, GOPPRR, GME, MS DSL Tools and MS Visio and tried to identify different categories, by grouping metamodel characteristics together. This work largely helped identify the different static features of this evaluation, such as the number of classes or relationships. In [LMWK14], the authors analyzed openly available UML models by using a set of various UML models and batch analyzing them by different metrics. The number of language units used per model and language unit usage frequency are calculated and evaluated, among other metrics. This previous work impacted this evaluation, which also uses countable metrics for the syntactic analysis since they can provide great insight into the static nature of metamodels before and after a transformation. Finally, the work of [DRDRIP14] introduced the concept of metamodeling measurement in general. This evaluation adopted the procedure described in this paper (i.e., metric calculation $\rightarrow$ defining metric correlation $\rightarrow$ selecting metric correlations $\rightarrow$ data analysis). Additionally, this paper further provided additional categories that could be used for static metamodel evaluation, such as number of generalizations and maximum inheritance level.

The evaluation is split in two parts: in subsection 6.1.1, the metrics for the transformation from ADOxx to EMF will be described and analyzed, in subsection 6.1.2 the metrics from EMF to ADOxx.

We gathered the metrics in various fashions: For gathering metrics in ADOxx, the evaluation procedure read an XML file containing the library information into an XML parser, analyzed the different attributes, and accumulated them. The source code is mainly used from the work of [Bor18], which analyzed metamodel metrics of ADOxx, accompanied by a Java XML analysis program. We slightly adapted the code to conform to the maven module structure that is used for this and other related modules of this thesis, but also to include additional metrics that were not present beforehand, like the total amount of enums used in a metamodel.

For gathering the metrics in Ecore, this evaluation is accompanied by a self-written standalone module that takes an `.ecore` file in XMI fashion as an input, loads the packages with the Ecore Java API [emfc], and iterates through all available elements in a programmatic way. This approach has the advantage of using the powerful Ecore API [emfc] with its language features in Java to easily navigate through the different elements, rather than using an XML parser that would have to include additional logic to achieve the same result. While iterating through the various elements, the procedure incremented a counter for every matching category and returned the results as a Java object containing all the different information.

The overall procedure consisted of taking a source metamodel from an environment, analyzing it, then transforming it, and analyzing the result. Finally, the result is outputed to an `.xlsx` file to provide adequate visual represent ability. The algorithm 6.1 gives a pseudo-code overview of the evaluation procedure.

Algorithm 6.1: Evaluation procedure of one analysis direction

```
  Input: (sourceMetamodel)
1 $sourceMetamodelMetrics \leftarrow analyzeMetamodel(sourceMetamodel)$
2 $targetMetamodel \leftarrow performMetamodelTransformation(sourceMetamodel)$
3 $targetMetamodelMetrics \leftarrow analyzeMetamodel(targetMetamodel)$
4 $evaluationExcelFile \leftarrow createEvaluationFile(sourceMetamodelMetrics, targetMetamodelMetrics)$
```

When evaluating either direction, we decided not to include the metamodel classes and attributes of ADOxx in the statistics. The abstract metamodel instances from ADOxx are irrelevant for the evaluation since they are a necessary, non-changeable, and equal part of every ADOxx metamodel. Not including these classes in the evaluation metrics shifts the focus to the actual metamodel transformation, which gives more expressiveness to the main research area of this work, which is to transform user-defined metamodels from one platform to another.

The static analysis consists of the following categories:

1. Successful import ratio in the target environment
2. Total number of classes per MM before - after
3. Total number of abstract classes per MM before - after
4. Total number of multi-inheritance classes per MM before - after
5. Total number of relationships per MM before - after
6. Total number of compositions per MM before - after
7. Total number of attributes per MM before - after
8. Total number of enumerations per MM before - after
9. Inheritance depth per MM before - after

6.1.1 ADOxx to EMF

In this chapter we evaluate and analyze the different metrics for each category defined in the previous section for the transformation direction ADOxx to EMF. Table 6.3 shows all the various source ADOxx metamodels that the evaluation procedure used in the experiment, along with their gathered metrics. In Table 6.4, the generated metamodels along with their metrics are represented.

Successful Import ratio In the transformation from ADOxx to EMF, the import success ratio is 100%, i.e., all transformed metamodels are importable to EMF. EMF does not have a complex import procedure like ADOxx; rather, the generated files can be opened in Eclipse with various tools, like the *Sample Ecore Model Editor*. As part of the evaluation procedure, we imported every `.ecore` file to the environment, which was error- and warning-checked. Errors in the metamodel would lead to displaying no metamodel information in the editor other then an error message; warnings in the metamodel occur, e.g., when duplicate names for attributes are used and manifest in the metamodel editor as a warning symbol on the affected entity. Viewing, opening and editing the metamodel is possible nonetheless. Those errors are categorized by the value *Errors/ warnings* in Table 6.2. The transformation produced valid files for every input metamodel without errors or warnings.

Table 6.2: Successful import metric for ADOxx to EMF transformation

Transformation Direction	Cases	No errors/ warnings	Errors/ warnings	Success rate
ADOxx → Ecore	45	45	0	100%

Classes before and after The size of classes before and after is the same for the source and the target environment for every metamodel. This finding validates that the transformation performs a 1 : 1 mapping of any class from the source to the target environment.

Abstract Classes before and after As with the classes, the abstract classes are equal for every metamodel in any environment. This finding validates that the transformation procedure correctly set every source abstract class to abstract in Ecore during the transformation process.

Multi-Inheritance Classes before and after The ADOxx metamodeling environment does not support multi-inheritance, thus the multi-inheritance count on ADOxx for every metamodel is zero. In EMF, the multi-inheritance count is also zero for every metamodel. This observation validates that no class falsely inherits from more than one class in the target environment.

Relationships before and after The relationships in ADOxx differ from those in the generated EMF metamodels. One important fact to notice is that the relationship size in EMF is always greater than or equals to in ADOxx, i.e. $|Rel_{emf}| \geq |Rel_{ado}|$. This stems from the fact that every `Interref` attribute IRC_{ado} in ADOxx is converted to an `EReference` Rel_{emf} in Ecore. In fact, the amount of relationships in Ecore corresponds to the sum of those interrefs and the normal relationships defined in the ADOxx metamodel $|Rel_{ado}|$. The complete formula thus is: $|Rel_{emf}| = |Rel_{ado}| + |IRC_{ado}|$. It is worth noting that the Interref count only applies to Interrefs that map to a class, and not to another model, as denoted by the letter C in the Interref notation. This validates the correct transformation of every `RelationClass` and `Interref` from ADOxx to EMF.

Compositions before and after In the metrics of Table 6.3, compositions are defined as every class that inherits from `__D_Container__`, while in EMF a composition is defined as a class that has a *containment* relationships to another class. Those values differ in either environment, showing that the transformation procedure does not yet map the compositions from an ADOxx source metamodel to compositions in EMF, as denoted by the value zero for every EMF metric in this category.

Attributes before and after Attribute sizes in ADOxx are always greater than their EMF counterpart, i.e. $|Att_{ado}| \geq |Att_{emf}|$. Since `Interref` attributes are always converted to an `EReference` and not to an attribute on Ecore side, the amount of attributes on Ecore side is always lower. The attributes count in EMF $|Att_{emf}|$ thus corresponds to $|Att_{emf}| = |Att_{ado}| - |IRC_{ado}|$, $|IRC_{ado}|$ being the amount of `Interrefs` that reference other classes in ADOxx. Through the comparative metrics, the expected behaviour of lower amounts of attribute on EMF side can be validated.

Enumerations before and after As can be seen in the metrics, the amount of enumerations in every source metamodel and their corresponding target metamodel is the same. This observation validates the correctness of the transformation part that is responsible for reading all enums from an ADOxx file and converting it to an `EEnum` on EMF side.

Inheritance depth before and after The inheritance level is for all metamodels of source and target platform equal. This finding resembles the desired behavior, where a distinct inheritance pattern from ADOxx is translated to the semantically equivalent inheritance pattern in EMF, thus validating the correctness of this transformation step.

Table 6.3: ADOxx source metamodels metrics

Metamodel	**Total Classes**	**Abstract Classes**	**Multi Inheritance Classes**	**Relationships**	**Compositions**	**Attributes**	**Enums**	**Inheritance Level**
4em	75	0	0	19	0	59	6	2
advisor	33	5	0	2	2	153	33	2
bd-ds	17	2	0	3	3	58	16	2
bpfm	58	12	0	36	9	664	170	3
bprim	18	1	0	11	0	2	2	1
codek	56	12	0	21	9	649	165	3
comvantage	71	2	0	21	7	243	45	4
cutide	180	24	0	54	9	581	262	6
diba	31	0	0	17	0	31	1	3
dice	72	12	0	22	10	31	10	5
dicer	14	0	0	4	1	49	10	3
dmn	9	2	0	4	2	52	2	2
e-gpm	37	2	0	31	5	347	66	2
enterknow	16	0	0	2	3	36	0	1
evaluationchains	52	0	0	29	3	265	73	3
fleet_mgmt_ado_export	8	3	0	3	0	15	1	2
fleet_mgmt_thesis	8	3	0	5	0	9	1	2
fleet_mgmt	7	3	0	2	0	11	1	2
hcm-l	12	0	0	9	2	207	5	3
horus	54	17	0	11	1	164	18	3
istar	14	1	0	10	1	57	17	3
jcs	11	0	0	1	0	12	10	3
kamet	23	1	0	6	2	16	2	2
kwd	84	4	0	31	9	800	196	3
learnpad	152	17	0	56	12	1168	270	3
local_env	11	2	0	2	0	15	3	2
melca	54	4	0	31	7	642	48	1
memo	127	3	0	81	0	740	105	2
muviemot	14	1	0	13	0	26	3	1
petrinets	6	2	0	7	0	5	1	1
pga	14	2	0	1	0	18	1	3
pss	55	0	0	21	9	86	37	3
public_transport_inf	9	4	0	3	0	16	1	2
rupert	98	0	0	26	13	839	176	4
save	22	0	0	10	4	49	13	3
sdbd	70	5	0	32	7	720	86	3
securetropos	28	0	0	22	2	85	1	3
semfis	96	4	0	39	13	1104	245	6
serm	5	0	0	3	0	7	1	1
simchronization	27	3	0	6	0	101	27	2
sIoT	116	0	0	26	4	912	262	4
smartcity	35	0	0	7	1	153	25	2
som	38	0	0	67	7	251	36	2
team	62	0	0	11	0	56	35	1
userstorymapping	13	0	0	5	1	115	20	2
min	5	0	0	1	0	2	0	1
median	29,5	2	0	11	2	85,5	17,5	3
max	180	24	0	81	13	1168	270	6

Table 6.4: Ecore transformed metamodels metrics

Metamodel	**Total Classes**	**Abstract Classes**	**Multi Inheritance Classes**	**Relationships**	**Compositions**	**Attributes**	**Enums**	**Inheritance Level**
4em	75	0	0	48	0	30	6	2
advisor	33	5	0	15	0	140	33	2
bd-ds	17	2	0	15	0	46	16	2
bpfm	58	12	0	43	0	657	170	3
bprim	18	1	0	5	0	8	2	1
codek	56	12	0	25	0	645	165	3
comvantage	71	2	0	94	0	170	45	4
cutide	180	24	0	33	0	602	262	6
diba	31	0	0	17	0	31	1	3
dicer	14	0	0	4	0	49	10	5
dicer	14	0	0	4	0	49	10	3
dmn	9	2	0	4	0	52	2	2
e-gpm	37	2	0	79	0	299	66	2
enterknow	16	0	0	18	0	20	0	1
evaluationchains	52	0	0	38	0	256	73	3
fleet_mgmt_ado_export	8	3	0	6	0	12	1	2
fleet_mgmt_thesis	8	3	0	5	0	9	1	2
fleet_mgmt	7	3	0	2	0	11	1	2
hcm-l	12	0	0	179	0	37	5	3
horus	54	17	0	26	0	149	18	3
istar	14	1	0	17	0	50	17	3
jcs	11	0	0	1	0	12	10	3
kamet	23	1	0	2	0	20	2	1
kwd	84	4	0	68	0	763	196	3
learnpad	152	17	0	165	0	1059	270	3
local_env	11	2	0	4	0	13	3	2
melca	54	4	0	101	0	572	48	1
memo	127	3	0	330	0	491	105	2
muviemot	14	1	0	17	0	22	3	1
petrinets	6	2	0	6	0	6	1	1
pga	14	2	0	1	0	18	1	3
pss	55	0	0	21	0	86	37	3
public_transport_inf	9	4	0	6	0	13	1	2
rupert	98	0	0	60	0	805	176	4
save	22	0	0	7	0	52	13	3
sdbd	70	5	0	105	0	647	86	3
securetropos	28	0	0	74	0	33	1	3
semfis	96	4	0	260	0	883	245	6
serm	5	0	0	3	0	7	1	1
simchronization	27	3	0	5	0	102	27	2
sIoT	116	0	0	88	0	850	262	4
smartcity	35	0	0	6	0	154	25	2
som	38	0	0	77	0	241	36	2
team	62	0	0	19	0	48	35	1
userstorymapping	13	0	0	16	0	104	20	2
min	5	0	0	1	0	6	0	1
median	27,5	1,5	0	17,5	0	52	17,5	3
max	180	24	0	330	0	1059	270	6

6.1.2 EMF to ADOxx

In this subsection we evaluate and analyze the different metrics for each category defined in the previous section for the transformation direction EMF to ADOxx. Table 6.6 shows all the various source Ecore metamodels that are used in the experiment along with their gathered metrics. In Table 6.7, the generated metamodels along with their metrics are represented.

Table 6.5: Successful import metric for EMF to ADOxx transformation

Transformation Direction	**Cases**	**No errors/ warnings**	**Errors/ warnings**	**Success rate**
EMF $\rightarrow$ ADOxx	33	32	1	96.97 %

Successful Import ratio When trying to import the generated metamodel to ADOxx, we successfully imported 32 out of the 33 metamodels, yielding a success rate of 96,97%. One metamodel could not be imported into ADOxx, since a `STRING` violation occurs on a certain attribute. In ADOxx, the maximum length of a `STRING` datatype value is limited to 3699 characters. When performing the transformation, the value of the *AttributeInterRefDomain* facet of the custom created attribute *composedInstances* matches a concatenated string of all the composed instances of this attribute, which conforms to the composed instances of the Ecore source metamodel. In this case, the composed instances are that many that the resulting string exceeds the length of 3699 characters, making it unable to import into ADOxx. The listing Code Example 6.1 shows a snippet of the affected attribute value. In a future iteration of the transformation procedure, we can mitigate this problem by storing exceeding strings in an external file and reading this file dynamically into the metamodel environment via *AdoScript*.

Classes before and after The size of classes for each metamodel in EMF and its counterpart in ADOxx is equal. This finding validates the correct behavior of the transformation, where a 1 : 1 for every class is defined. It further shows that the transformation procedure did not omit any class or falsely created additional classes during this process.

Abstract Classes before and after As with the classes, the amount of abstract classes is equal in EMF and in ADOxx, showing that the *ClassAbstract* attribute on ADOxx side is correctly assigned to the appropriate classes.

Multi-Inheritance Classes before and after A few of the source Ecore metamodels have multiple inheritance pattern incorporated. On ADOxx, the expected value is zero for all metamodels since no multi-inheritance is possible in this platform.

```
FACET <AttributeInterRefDomain>
VALUE "REFDOMAIN
OBJREF
    mt:\"KML\"
    c:\"Message\"
    min:0
OBJREF
    mt:\"KML\"
    c:\"BalloonStyle\"
    min:0
OBJREF
    mt:\"KML\"
    c:\"ViewRefreshTime\"
    min:0
...
```

Code Example 6.1: Affected AttributeInterRefDomain value causing import to fail

Relationships before and after Relationships in EMF are always greater or equal in size than in ADOxx, i.e. $|Rel_{emf}| \geq |Rel_{ado}|$. Since compositions in Ecore are represented as `EReferences` (just with an additional flag set), the number of relationships is expected to be higher in EMF. In ADOxx, the composition feature is realized by using certain attributes and an additional AdoScript part, which this work described in subsection 5.3.2. Since those compositions in Ecore Com_{emf} are not represented in ADOxx, the size of relationships in ADOxx $|Rel_{ado}|$ should correspond to the following formula: $|Rel_{ado}| = |Rel_{emf}| - |Com_{emf}|$. This formula is valid for all tuples of metamodels, yielding a correct syntactic behavior of the transformation procedure.

Compositions before and after By default, ADOxx defines compositions by any class that inherits from the abstract metamodel class `__D_Container__`. Since the transformation procedure used a different approach for this transformation direction, as described in subsection 5.3.2, the it should have created no composition on ADOxx side, i.e., it should generate no class, that inherits from the class `__D_Container__`. We can validate this behavior through the values of *composition* on the transformed ADOxx metrics, which are all equal to zero.

Attributes before and after The number of attributes in the transformed ADOxx metamodels is expected to be higher than or equal to those of their corresponding Ecore source metamodels, i.e., $|Att_{emf}| \leq |Att_{ado}|$. This observation stems from the fact that a $1:1$ mapping of attributes exist for this direction, but the transformation procedure also creates several other attributes in the transformation process. For one, the three Library attributes *compositeClasses*, *compositumClasses* and *multiInheritanceClasses* are included in this metric. These attributes ($|Att_{meta}|$) are created in the transformation process and contain various information about the multi-inheritance and composition patterns. Additionally, for every composition on the EMF side, a corresponding attribute named *compositumClass* is created on the corresponding classes on ADOxx side, as well as an attribute *validIncomingRelationships* for classes that are affected by multiple inheritance and incoming relationships. We can not derive a general formula for the number of attributes on the target platform without internal knowledge of the concrete inheritance hierarchy since it strongly depends on the structure of the source metamodel (e.g., when a superclass is also part of a composition, the attribute *compositumClass* is edited in the superclass and removed from the subclass, which is still valid for all subclasses; this knowledge is only derivable when analyzing the structure of the metamodel). However, we can verify the expected behavior of having fewer attributes than in the target metamodel for every metamodel used in the experiments.

Enumerations before and after The count of enumerations on both sides are equal, yielding the correct behaviour of the transformation, where a $1:1$ mapping occurs between enumerations from ADOxx to EMF.

Inheritance depth before and after When the transformation procedure translates multiple inheritance patterns to single inheritance patterns, the inheritance depth may decrease in the target metamodel. The procedure can create new inheritance paths, where classes inherit from other classes of a smaller branch while the larger branch stops. The inheritance depth cannot be higher, though, since this would yield the fact that the transformation procedure created additional classes in the transformation process. Since there is a $1:1$ mapping of the classes in this direction, this can never happen.

Generally speaking, evaluating the correct inheritance procedure can be performed by checking the following formula $ID_{emf} \geq ID_{adoxx}$, where ID stands for *inheritance depth*. We can validate this behavior for all metamodels in the experiment.

Table 6.6: Ecore source metamodels metrics

Metamodel	**Total Classes**	**Abstract Classes**	**Multi Inheritance Classes**	**Relationships**	**Compositions**	**Attributes**	**Enums**	**Inheritance Level**
bdd	8	2	0	14	5	3	0	1
bibtex_mod	14	6	4	2	2	8	1	3
bibtex	14	6	4	2	2	7	0	3
book	2	0	0	2	1	4	0	1
books	2	0	0	1	1	3	0	1
bugzilla	9	1	0	8	7	35	7	1
cpl	32	6	1	16	16	42	0	4
dot	26	8	0	30	11	44	0	4
dsl	13	5	0	13	5	14	0	4
dslmodel	14	3	0	15	7	8	0	2
fleet_management	8	3	1	3	1	8	1	2
km2	14	4	0	9	5	8	0	3
km3	15	2	0	17	6	10	0	4
kml	93	4	0	2	1	64	2	3
metrics	4	1	0	0	0	5	0	1
mmelementlist	3	1	0	1	1	1	0	1
mmtree	3	1	0	1	1	2	1	1
mysql	8	1	0	7	4	7	0	2
persons	3	1	0	0	0	1	0	1
problem	1	0	0	0	0	3	1	1
pub	1	0	0	0	0	2	0	1
rss	9	0	0	16	9	45	0	1
sample	4	2	1	0	0	4	2	1
simplerdbms	3	0	0	5	2	3	0	1
table	3	0	0	2	2	1	0	1
tcs	66	12	0	59	36	54	4	3
textualpathexp	5	1	0	3	3	2	0	1
typea	2	0	0	2	2	1	0	1
typeb	2	0	0	1	1	1	0	1
xml	5	1	0	2	1	6	0	2
min	1	0	0	0	0	1	0	1
median	6.5	1	0	2	2	5.5	0	1
max	93	12	4	59	36	64	7	4

Table 6.7: ADOxx transformed metamodels metrics

Metamodel	**Total Classes**	**Abstract Classes**	**Multi Inheritance Classes**	**Relationships**	**Compositions**	**Attributes**	**Enums**	**Inheritance Level**
bdd	8	2	0	9	0	14	0	1
bibtex_mod	14	6	0	0	0	25	1	3
bibtex	14	6	0	0	0	23	0	3
book	2	0	0	1	0	9	0	1
books	2	0	0	0	0	8	0	1
bugzilla	9	1	0	1	0	47	7	1
cpl	32	6	0	0	0	74	0	4
dot	26	8	0	19	0	64	0	4
dsl	13	5	0	8	0	27	0	4
dslmodel	14	3	0	8	0	26	0	2
fleet_management	8	3	0	3	0	18	1	2
km2	14	4	0	4	0	25	0	3
km3	15	2	0	11	0	26	0	4
kml[1]	-	-	-	-	-	-	-	-
metrics	4	1	0	0	0	8	0	1
mmelementlist	3	1	0	0	0	6	0	1
mmtree	3	1	0	0	0	8	1	1
mysql	8	1	0	3	0	19	0	2
persons	3	1	0	0	0	4	0	1
problem	1	0	0	0	0	6	1	1
pub	1	0	0	0	0	5	0	1
rss	9	0	0	7	0	59	0	1
sample	4	2	0	0	0	8	2	1
simplerdbms	3	0	0	3	0	9	0	1
table	3	0	0	0	0	8	0	1
tcs	66	12	0	23	0	132	4	3
textualpathexp	5	1	0	0	0	11	0	1
typea	2	0	0	0	0	6	0	1
typeb	2	0	0	0	0	6	0	1
xml	5	1	0	1	0	14	0	2
min	1	0	0	0	0	4	0	1
median	5	1	0	0	0	14	0	1
max	66	12	0	23	0	132	7	4

[1] Metamodel transformation infeasible

6.2 Semantic Evaluation

To test the semantic correctness of the generated metamodels, we conducted a semantic evaluation. The evaluation is split in two parts: in subsection 6.2.1, the semantic evaluation for the transformation from ADOxx to EMF will be described and analyzed, in subsection 6.2.2 the semantic evaluation from EMF to ADOxx.

The semantic evaluation differs from the syntactic evaluation described in section 6.1, as we needed a manual procedure to test metamodel features. With the semantic evaluation, we should collect results about the correct behavior of the generated metamodels. The evaluation gathers insights about the semantic information heterogeneity, as described in section 2.4, which plays a vital role in metamodel interoperability. Only if two metamodels from a source and target platform behave semantically correct, then the transformation can be described as an interoperability mechanism.

The procedure for creating the evaluation looks as follows:

1. Transform a source metamodel into a target metamodel
2. Import the target metamodel into the target platform
3. Create a model instance from the generated metamodel
4. Create different elements based on categories described below
5. Evaluate the correct behavior

The transformation procedure uses different categories to validate the correct behavior. These categories should form a basis for a correct evaluation of semantic behavior between the two platforms. The categories are listed in the following and described in the paragraph below:

1. Correctly mapped classes
2. Correctly mapped relationships
3. Correctly mapped attributes (including enums)
4. Correctly mapped multiple inheritance patterns
5. Correctly mapped composition patterns

The categories of *Correctly mapped classes* and *Correctly mapped relationships* were chosen to show that classes and relationships can be defined in the target metamodel platform by the same semantic rules as in in the source metamodel. I.e., during the evaluation, we created classes in the target environment and compared their types and names, showing that they can be added to the target model instance and are correctly mapped. Relationships are added between various classes, showing that a user can only draw certain relationships between the two classes defined in the source metamodel. Also, if a relationship is created in the source metamodel with an abstract class as its target, it is checked in the target model instance that only the subclasses of this abstract class are referenceable through that relationship.

The category of *Correctly mapped attributes* verifies that a user can create attributes with the corresponding type and name in the target platform as defined in the source platform. For ADOxx, we also checked that the attribute representation is added correctly to the metamodel classes by checking that the popup of a certain class instance contains all the possible attributes as defined in the source metamodel. Additionally, enums are also checked in this step and evaluated in this category since they are a special type of attribute strongly connected to attributes both in ADOxx and in EMF. For enums, we also checked the correct setting of the default value when instantiating an enum instance, showing that the transformation procedure performed the mapping correctly. Furthermore, we checked that an attribute of `enumlist` of ADOxx, which yields multi-valued enums, is correctly set on Ecore side as `Enums` with a higher upper bound and vice versa.

The category *Correctly mapped multiple inheritance pattern* is only evaluated in the direction of ADOxx to EMF since only Ecore supports multiple inheritance. The procedure as described in the paragraph Process Classes With Multiple Inheritance is evaluated here. For this purpose, the evaluation checks that the non-inherited classes in ADOxx contain all other attributes that used to be part of the multi-inheritance class in Ecore. Also, the procedure evaluates the correct behavior of the incoming relationships, meaning that the newly created relationships that have `__D_construct__` as their target type are only validly drawable to the same classes as defined in the Ecore source metamodel.

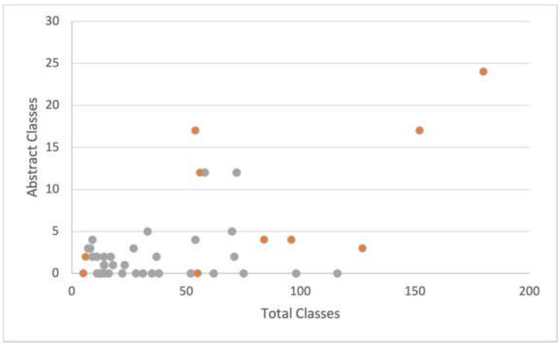
Finally, the category *Correctly mapped composition patterns* is evaluated. While no composition classes as defined in ADOxx are mapped as compositions in Ecore, compositions are defined during the transformation from ADOxx to Ecore as described in the paragraph Add basic containments. For this direction, we, therefore, evaluated that a user can correctly create basic containment compositions on the EMF side, i.e., any element can only be a child of the class `RootClass` and creatable through either the composition *dynamicContainments* or *staticContainments*. For the direction from Ecore to ADOxx, we checked those composite elements, as defined in the Ecore source metamodel, are only creatable when an instance of the composite element exists (i.e., the AdoScript behaves correctly and shows a popup with the different class instances that are possibly assignable as the compositum class). Additionally, we checked the correct behavior when deleting a compositum class, which should cause ADOxx to delete all composite elements, as defined in the AdoScript written for this purpose.

Since the evaluation of these metrics has to be performed manually, we decided to choose a representable subset of ten metamodels per direction. The reason is mainly that the manual evaluation is very time-consuming, and testing all the different combinations by hand imposes great effort on the person evaluating. As will be described in subsection 6.2.1

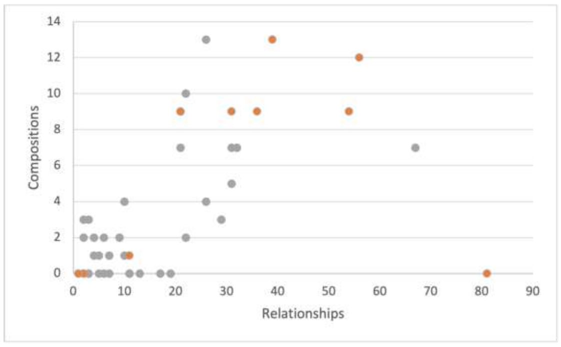
and subsection 6.2.2, the metamodels per source environment are chosen based on the different categories as introduced and described in section 6.1. We tried to include metamodels of different sizes per category to have many metamodels with different characteristics. This way, the evaluation results could be generalized, stating that the transformation behaves on a certain level on source metamodels of any size of characteristics.

6.2.1 ADOxx to EMF

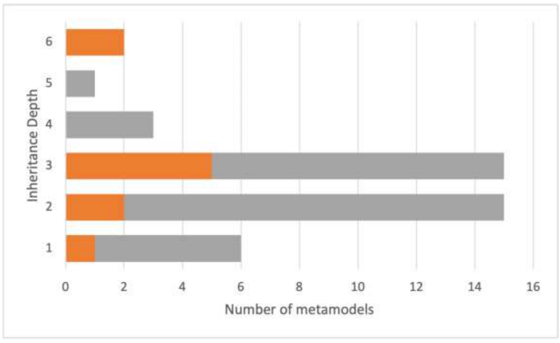
Figure 6.1 gives an overview of the distribution by tuples of metric categories for all the metamodels of the experiments. The orange dots in the graphics resemble the chosen metamodels for the semantic evaluation. In general it was tried to create an even distribution that covers metamodels of small, large and medium characteristic size.



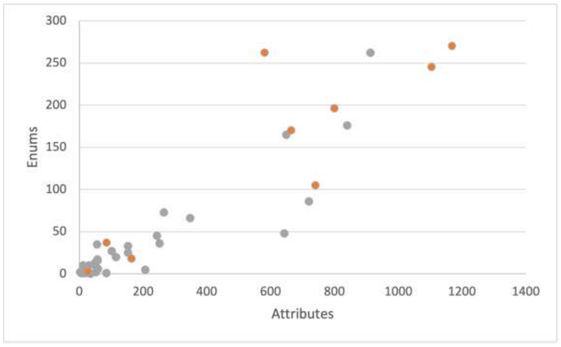
(a) Total classes and abstract class distribution



(b) Relationships and Compositions distribution



(c) Inheritance depth distribution



(d) Attributes and Enums distribution

Figure 6.1: Overall distribution and selected metamodels of the ADOxx data set

The distribution of classes and abstract classes is fairly even, including both the smallest, the largest and several metamodel in between by those category sizes.

For relationships, both the metamodels with the highest and lowest amount of relationships, as well as the highest and lowest amount of ADOxx compositions were chosen.

Several metamodels in between were chosen, while focusing on a greater variety of relationships count rather than composition counts, since ADOxx compositions are not part of the transformation of this direction.

We chose metamodels of both the lowest and highest depth for inheritance depth. Several metamodels in between were chosen as well, matching the distribution of the dataset, i.e., choosing metamodels that have an inheritance depth of two or three.

Finally, the attributes and enums categories distribution shows that this evaluation uses metamodels with the highest amount of attributes and enums. Several metamodels were chosen, where the respective count is comparably low, but the occurrence, as seen in the distribution chart, is fairly high. In general, we tried to cover a greater variety of both attributes and enums since both of these concepts are part of this direction's transformation.

Table 6.8: Result of semantic evaluation for transformation from ADOxx to EMF

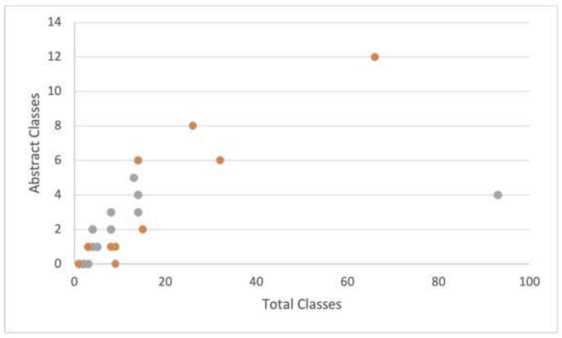
	Class functionality	**Relationship functionality**	**Attribute functionality**	**Multiple inheritance functionality**	**Composition functionality**
bibtex	✓	✓	✓	-	✓
bugzilla	✓	✓	✓	-	✓
cpl	✓	✓	✓	-	✓
dot	✓	✓	✓	-	✓
km3	✓	✓	✓	-	✓
mmelementlist	✓	✓	✓	-	✓
mysql	✓	✓	✓	-	✓
problem	✓	✓	✓	-	✓
rss	✓	✓	✓	-	✓
tcs	✓	✓	✓	-	✓

As can be seen in Table 6.8, the semantic evaluation performed positively in all cases for every category of every chosen metamodel. This finding validates the semantic correctness of the transformation in this direction for the chosen metamodels.

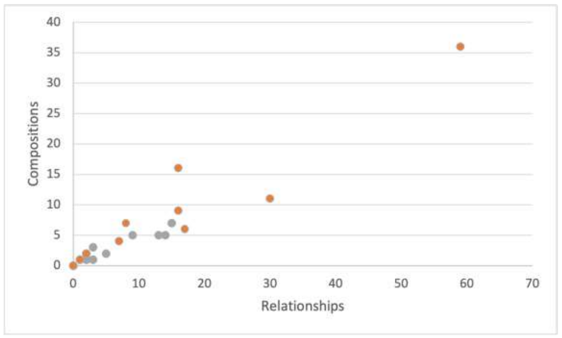
While a greater amount of metamodel analysis in a semantic context would be needed to make assumptions of semantic correctness for an arbitrary set of metamodels, the variety with which the metamodels we chose from the dataset can make first general assumptions about the correct behavior of this transformation direction.

6.2.2 EMF to ADOxx

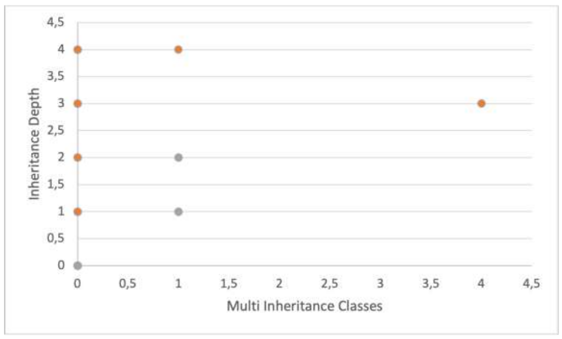
Figure 6.2 gives an overview of the distribution of the different metamodels of the dataset, represented as tuples by metric categories. The orange dots in the matrices represent the chosen metamodels for this evaluation. It was again tried to cover a great variety of metamodels for the semantic evaluation, which can be interpreted as a fair distribution of chosen metamodels based on the overall distribution.



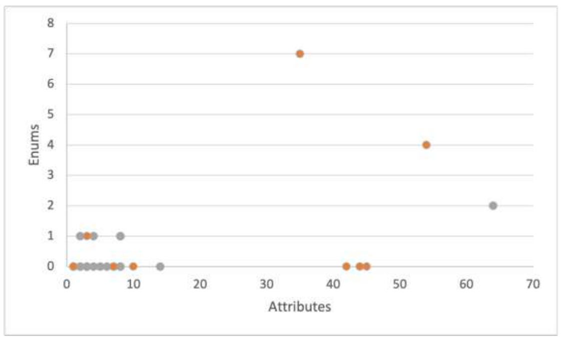
(a) Total classes and abstract class distribution



(b) Relationships and Compositions distribution



(c) Inheritance distribution



(d) Attributes and Enums distribution

Figure 6.2: Overall distribution and selected metamodels of the Ecore data set

For classes and abstract classes it was tried to include the metamodels with the lowest and the highest of these categories. Since it is not possible to import one metamodel to ADOxx, as described in subsection 6.1.2, this particular metamodel was not chosen for this evaluation. This metamodel has the highest count of classes in the dataset; it was dropped for the metamodel with the next highest class count, which also happens to be the metamodel with the highest abstract class count. The rest of the metamodels were chosen in order to represent a fair distribution of the remaining elements.

As for relationships and compositions, the evaluation procedure uses metamodels of min and max values for these categories. Various metamodels in between were chosen, creating a fair distribution of these remaining elements.

For the inheritance distribution, the evaluation focused on covering metamodels with different inheritance depths, as this reveals the correct behavior of the transformation algorithm handling inheritance. Another key aspect lies in choosing metamodels with a high count of multiple inheritance classes, as this reveals the correct behavior of the algorithm that transforms multi-inheritance classes to normal classes with additional attributes and correct incoming relationship behavior in ADOxx.

Finally, the evaluation procedure uses metamodels with various attributes and enums. Again, it is not possible to use the metamodel with the highest attribute count since a user cannot import this metamodel to ADOxx, as described in subsection 6.1.2. We chose the metamodel with the next highest count of attributes instead. Besides that, the evaluation included metamodels with the highest enum count and the lowest attribute and enum count. While there is not a lot of variety in enum sizes in the dataset, as can be seen by the flat distribution of most metamodels on the Y-Axis, the focus shifted to having a fair distribution of chosen metamodels based on their attribute count.

Table 6.9: Result of semantic evaluation for transformation from EMF to ADOxx

	Class functionality	**Relationship functionality**	**Attribute functionality**	**Multiple inheritance functionality**	**Composition functionality**
bpfm	✓	✓	✓	✓	✓
cutide	✓	✓	✓	✓	✓
fleet_management	✓	✓	✓	✓	✓
horus	✓	✓	✓	✓	✓
kwd	✓	✓	✓	✓	✓
learnpad	✓	✓	✓	✓	✓
memo	✓	✓	✓	✓	✓
pss	✓	✓	✓	✓	✓
semfis	✓	✓	✓	✓	✓
serm	✓	✓	✓	✓	✓

Table 6.9 shows the results of the semantic evaluation of this transformation's direction. As can be seen, the evaluation is valid for all instances of the used metamodels in all categories, validating the correct semantic behavior of these metamodel transformations based on the implemented solution.

While this small subset is not representative enough to make conclusions for an arbitrary set of metamodels, we can make general assumptions about the successful semantic transformation of metamodels since the evaluation used a great variety of metamodels with different characteristics in the experiments.

CHAPTER 7

Future Work

While this work concluded with the interoperability analysis on metamodel level, several other areas of research are not part of this scientific work. However, this work could function as a basis for further analysis and research in related areas.

The remaining metamodel features that are not part of this scientific work can be analyzed and implemented. This research direction includes an analysis of the feasibility of transforming the constraint language features from one platform to another but also transforming compositions defined in ADOxx by the abstract class `__D_container__` to compositions in the Ecore target metamodel. Implementing the mapping of record classes could be another research area, for which a mapping approach was already described in this thesis, but was not yet implemented.

For the other, researchers could conduct a transformation on model level. While this work solely focused on metamodel transformation, the generated insights can be used on a model transformation from ADOxx to Ecore and vice versa. Since this work shows the feasibility of successfully transforming metamodels and the fact that models always conform to a certain metamodel, an additional source model file could be provided together with the metamodel file to transform both artifacts. A proposed solution is to create a mapping table on the metamodel transformation step that maps the names of the first-class concepts from the source platform to the target platform. This mapping table is needed since the metamodel transformation procedure might rename first-class concepts during the transformation. Taking a source model file and loading it with the context of the metamodel file would enable creating a valid model file on the target platform programmatically. For ADOxx, researchers could write a string parser, which maps the model elements from an EMF model to an `.adl` file, which is a human-readable format of a model definition and which can be imported into ADOxx [adob]. On Ecore side, one could use the *Ecore-Reflection API*, which is part of the Ecore Java API [emfc]. Through this API, the different model elements could be created reflectively, enabling model transformation also on Ecore side.

Another research area would be to transform the concrete syntax of the metamodel and model elements to the target platform. Both ADOxx and EMF allow the definition of graphical elements accompanied by a certain metamodel. In ADOxx, a transformation procedure could create graphical representations within a *GraphRep* attribute of a class, which is specifiable by a distinct syntax of this ADOxx attribute. But also the definition of a graph representation in `.svg` syntax would be a possible solution to create the concrete syntax. On Ecore side, plugins like Sirius [emfd] could be used, that allow to define concrete syntax in `.svg` as well. This approach would be a trivial and feasible transformation of the concrete syntax since `.svg` files are enabled in both platforms by default.

Finally, one could go further than the meta-meta model level usually present in model-driven engineering. As was shown in this scientific work, some metamodeling concepts differ a lot from other metamodeling concepts. E.g., in ADOxx, only single inheritance is supported, while in EMF, multiple inheritance is possible for user-defined classes. An in-depth research area would be to define a 4th model layer on top of the meta-meta model layer (the meta-meta-metamodel layer), which would describe all the meta-metamodel features like inheritance, composition, and so forth present in a metamodel environment. The scientific work would then include the research of defining general mappings of those features on this particular level, which could benefit model transformation for arbitrary metamodeling platforms.

CHAPTER 8

Conclusio

This scientific work focused on creating an M3 level based bridge that would enable interoperability between the two metamodeling platforms ADOxx and EMF.

It has done so by introducing the concepts of metamodeling and interoperability and then introducing the concrete technologies of ADOxx and EMF. It further revealed the state-of-the-art of metamodel platform interoperability, showing that several solutions are already implemented for a variety of metamodeling platforms, yet no scientific mention of creating interoperability between ADOxx and EMF is present. A comparison of the two metamodeling platforms that followed revealed that both platforms share many similarities, like their first-class concepts, but also have a lot of differences, like their inheritance multiplicity. The next part of this work focused on describing the proposed M3 level bridges' concrete implementation details based on the previous steps' comparison results. Those bridges include a transformation procedure for ADOxx metamodels to EMF metamodels and a procedure for transforming EMF metamodels to ADOxx metamodels. Finally, this work contains an experiment with a distinct set of metamodels of either platform, which included executing the transformation procedures and collecting static data (like class counts) and semantic data (like correct behavior of composition patterns). We investigated the results in a context of syntactic correctness and semantic equivalence. This work showed that most metamodels can be transformed into a metamodel of the target platform. Their static features are syntactically valid, and their behavior on the target platform is semantically equivalent to the source platform. We also showed that failing imports in the current iteration of the transformation procedure can be mitigated in the future through implementing special procedures, increasing the amount of importable metamodels into the target platform ADOxx., At the beginning of this work, two research questions were introduced. They are mentioned here again, followed by their answers that this thesis provided:

1. **Is interoperability between the two metamodel platforms feasible?**
   Yes, interoperability is feasible for most metamodels. The experiments showed that the transformation of ADOxx metamodels to EMF metamodels is always feasible. On the other hand, this work showed that certain metamodel elements might be too large for some data types in ADOxx. This finding poses a certain limitation to the transformation feasibility. A proposed solution of reading the affected values dynamically through *AdoScript* can mitigate this problem though, which bypasses the limitation of data definition boundaries.

   Since these transformations allow the import of target metamodels to the target environment, interoperability of the platform is proven, according to the interoperability definitions in section 2.4.

2. **How well performs the proposed solution syntactically and semantically?**

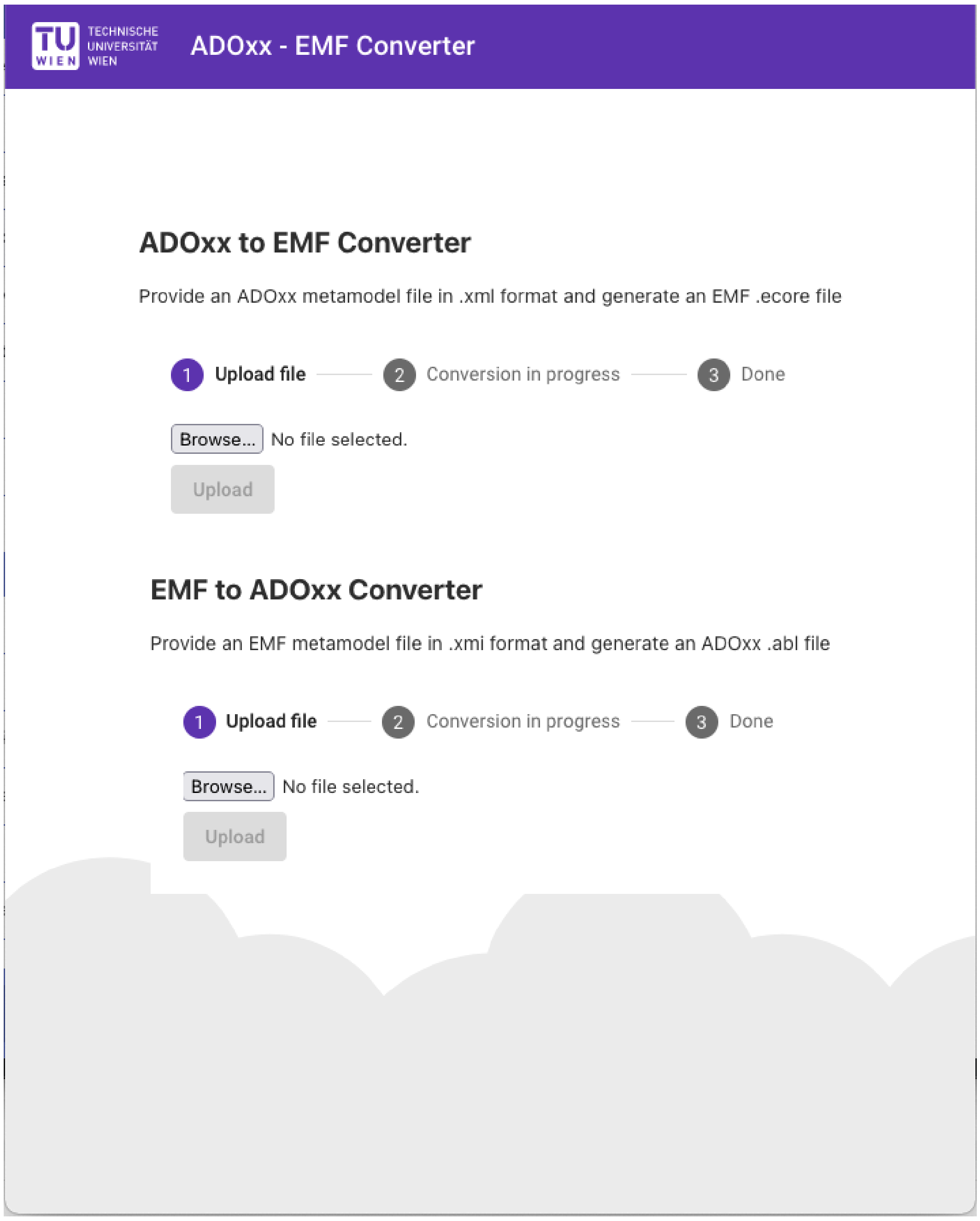
   The evaluation showed that for the metamodels that could be transformed behaved syntactically correct and semantically equivalent.

   In section 6.1 the evaluation showed that the gathered metrics for the transformed metamodels are equivalent to the expected metrics for a certain category. It also showed that those transformed metamodels could be imported and used on either platform without yielding any errors or warnings. This observation shows that the transformation behaved expectedly and the transformation procedure achieved syntactic correctness of the transformed files.

   In section 6.2 the evaluation showed that the semantic behavior is equivalent in source and target platforms for both transformations. We could verify simple semantic features, like the class and attribute creation being equal on either platform. For the transformation direction from Ecore to ADOxx, We could verify that complex semantic features, like the correct behavior of compositions and multi-inheritance classes, perform semantically equivalent as they did in the source environment of EMF.

This work concludes with possible future research areas in chapter 7 that build upon this thesis's topic and can yield great benefit to the domain of Model-Driven Engineering.

The transformation procedure will be deployed openly as a web app which is navigateable through the URL `http://me.big.tuwien.ac.at/`. A user can open this site on a web browser and perform the unidirectional transformations by providing a source metamodel of the platforms ADOxx or EMF. Figure 8.1 shows a screenshot of the so-far developed web application.


ADOxx - EMF Converter

ADOxx to EMF Converter

Provide an ADOxx metamodel file in .xml format and generate an EMF .ecore file

EMF to ADOxx Converter

Provide an EMF metamodel file in .xmi format and generate an ADOxx .abl file

Figure 8.1: Screenshot of the deployed transformation web app

List of Figures

List of Tables

List of Algorithms

List of Code Examples

Bibliography

[adoa] The adoxx metamodelling platform - welcome to adoxx.org - adoxx.org. `https://www.adoxx.org/live/home`. Accessed: 2022-04-01.

[adob] Adoxx model language adl - adoxx.org. `citehttps://www.adoxx.org/live/model-language-adl`. Accessed: 2022-04-01.

[adoc] Adoxx_all_api_public - gitlab. `https://git.boc-group.eu/adoxx/adoxx_all_api_public`. Accessed: 2022-04-01.

[adod] All2abl converter service - adoxx.org. `https://www.adoxx.org/live/all2abl-converter-service`. Accessed: 2022-04-01.

[adoe] Attribute facets - adoxx.org. `https://www.adoxx.org/live/facets`. Accessed: 2022-04-01.

[adof] Class attribute and attribute - adoxx.org. `https://www.adoxx.org/live/class-attribute-and-attribute`. Accessed: 2022-04-01.

[adog] Predefined abstract classes (dynamic) - adoxx.org. `https://www.adoxx.org/live/predefined-abstract-classes-dynamic-`. Accessed: 2022-04-01.

[adoh] Scripting language adoscript - adoxx.org. `https://www.adoxx.org/live/scripting-language-adoscript`. Accessed: 2022-04-01.

[BBK+19] Dominik Bork, Robert Andrei Buchmann, Dimitris Karagiannis, Moonkun Lee, and Elena-Teodora Miron. An open platform for modeling method conceptualization: The omilab digital ecosystem. *Commun. Assoc. Inf. Syst.*, 44:32, 2019.

[BCC+10] Hugo Brunelière, Jordi Cabot, Cauê Clasen, Frédéric Jouault, and Jean Bézivin. Towards Model Driven Tool Interoperability: Bridging Eclipse and Microsoft Modeling Tools. In *Modelling Foundations and Applications - 6th European Conference, ECMFA 2010, Proceedings*, pages 32–47. Springer, 2010.

[BCWB17] Marco Brambilla, Jordi Cabot, Manuel Wimmer, and Luciano Baresi. *Model-Driven Software Engineering in Practice : Second Edition*. Morgan & Claypool Publishers, 2017.

[BG14] Fabian Büttner and Martin Gogolla. On ocl-based imperative languages. *Science of Computer Programming*, 92:162–178, 2014. Selected papers from the Brazilian Symposium on Formal Methods (SBMF 2011).

[BKP20] Dominik Bork, Dimitris Karagiannis, and Benedikt Pittl. A survey of modeling language specification techniques. *Inf. Syst.*, 87, 2020.

[Bor18] Dominik Bork. Metamodel-based analysis of domain-specific conceptual modeling methods. In Robert Andrei Buchmann, Dimitris Karagiannis, and Marite Kirikova, editors, *The Practice of Enterprise Modeling*, pages 172–187, Cham, 2018. Springer International Publishing.

[BPB21] Ilia Bider, Erik Perjons, and Dominik Bork. Towards on-the-fly creation of modeling language jargons. In Vadim Ermolayev, David Esteban, Heinrich C. Mayr, Mykola Nikitchenko, Sergiy Bogomolov, Grygoriy Zholtkevych, Vitaliy Yakovyna, and Aleksander Spivakovsky, editors, *Proceedings of the 17th International Conference on ICT in Education, Research and Industrial Applications. Integration, Harmonization and Knowledge Transfer. Volume I: Main Conference, PhD Symposium, and Posters, Kherson, Ukraine, September 28 - October 2, 2021*, volume 3013 of *CEUR Workshop Proceedings*, pages 142–157. CEUR-WS.org, 2021.

[BWA21] Dominik Bork, Manuel Wimmer, and Konstantinos Anagnostou. Towards interoperable metamodeling platforms: The case of bridging adoxx and emf. 2021.

[CG12] Jordi Cabot and Martin Gogolla. Object constraint language (OCL): A definitive guide. In *Formal Methods for Model-Driven Engineering*, Lecture notes in computer science, pages 58–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[CGB+21] Alessandro Colantoni, Antonio Garmendia, Luca Berardinelli, Manuel Wimmer, and Johannes Bräuer. Leveraging model-driven technologies for JSON artefacts: The shipyard case study. In *24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 250–260. IEEE, 2021.

[CMR02] Yania Crespo, José Marqués, and Juan Rodríguez. On the translation of multiple inheritance hierarchies into single inheritance hierarchies. pages 30–37, 01 2002.

[cs.] http://www.cs.sjsu.edu/ pearce/modules/lectures/uml2/index.htm). http://www.cs.sjsu.edu/~pearce/modules/lectures/uml2/index.htm. Accessed: 2022-04-01.

[DRDRIP14] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Mining metrics for understanding metamodel characteristics. In *Proceedings of the 6th International Workshop on Modeling in Software Engineering - MiSE 2014*, New York, New York, USA, 2014. ACM Press.

[emfa] Atl | the eclipse foundation. https://www.eclipse.org/atl/. Accessed: 2022-04-01.

[emfb] Atl transformations | the eclipse foundation. https://www.eclipse.org/atl/atlTransformations/. Accessed: 2022-04-01.

[emfc] Overview (emf javadoc). https://download.eclipse.org/modeling/emf/emf/javadoc/2.4.2/overview-summary.html. Accessed: 2022-04-01.

[emfd] Sirius | home. https://www.eclipse.org/sirius/. Accessed: 2022-04-01.

[FRK06] Hans-Georg Fill, Timothy Redmond, and Dimitris Karagiannis. FDMM: A Formalism for Describing ADOxx Meta Models and Models. 2006.

[Gro15] Object Management Group. Xml metadata interchange (xmi) specification. Technical report, Object Management Group, 2015.

[jso] Convert xml to java object online - json2csharp toolkit. https://json2csharp.com/xml-to-java. Accessed: 2022-04-01.

[KBJK03] Harald Kühn, Franz Bayer, Stefan Junginger, and Dimitris Karagiannis. D.: „enterprise model integration. volume 2738, pages 379–392, 09 2003.

[Ker08] Heiko Kern. The Interchange of (Meta)Models between MetaEdit+ and Eclipse EMF Using M3-Level-Based Bridges. 2008.

[Ker16] Heiko Kern. *Model interoperability between meta-modeling environments by using m3-level-based bridges.* PhD thesis, Leipzig University, Germany, 2016.

[KHK11] Heiko Kern, Axel Hummel, and Stefan Kühne. Towards a comparative analysis of meta-metamodels. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, pages 7–12. ACM, 2011.

[KK07] Heiko Kern and Stefan Kühne. Model Interchange between ARIS and Eclipse EMF. In *7th OOPSLA Workshop on Domain-Specific Modeling at OOPSLA*, volume 2007, 2007.

[KM06] Harald Kühn and Marion Murzek. Interoperability Issues in Metamodelling Platforms. 2006.

[LMWK14] Phillip Langer, Tanja Mayerhofer, Manuel Wimmer, and Gerti Kappel. On the usage of uml: On the usage of uml. In Hans-Georg Fill, Dimitris Karagiannis, and Ulrich Reimer, editors, *Modellierung 2014*, pages 289–304. GI, 2014.

[NBM$^+$15] Patrick Neubauer, Alexander Bergmayr, Tanja Mayerhofer, Javier Troya, and Manuel Wimmer. XMLText: from XML schema to xtext. In *Proc. of the ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, pages 71–76. ACM, 2015.

[PRTV12] Ken Peffers, Marcus A. Rothenberger, Tuure Tuunanen, and Reza Vaezi. Design science research evaluation. In Ken Peffers, Marcus A. Rothenberger, and William L. Kuechler Jr., editors, *Design Science Research in Information Systems. Advances in Theory and Practice - 7th International Conference, DESRIST 2012, Las Vegas, NV, USA, May 14-15, 2012. Proceedings*, volume 7286 of *Lecture Notes in Computer Science*, pages 398–410. Springer, 2012.

[RBKJ06] Jolita Ralyte, Per Backlund, Harald Kühn, and Manfred Jeusfeld. Method chunks for interoperability. volume 4215, pages 339–353, 11 2006.

[WK05] Manuel Wimmer and Gerhard Kramler. Bridging grammarware and modelware. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, pages 159–168. Springer, 2005.

Appendix: DevOps Manual

This DevOps manual explains the structure of the transformation program, the required prerequisites to execute any program, and finally, an explanation of the execution possibilities implemented in this solution.

Module introduction

The overall source code consists of different Maven modules, which shall ease the use of the encapsulated functionality of the program for any specific use case.

The module `adoxx-ecore-converter` is the central module of this application. It is the root of all the other modules that are part of this environment. A structure of the module hierarchy is displayed in the following. An explanation of the different modules follows further below:

- `adoxx-ecore-converter`
  - `adoxx-all-api-public`
  - `converter-shared`
  - `adoxx-to-ecore-converter`
  - `ecore-to-adoxx-converter`
  - `adoxx-evaluator`
  - `ecore-evaluator`
  - `executor`
  - `adoxx-ecore-rest-controller`
  - `adoxx-ecore-converter-frontend` *(not part of Maven modules)*

adoxx-ecore-converter

This module is the main module of the whole development project. It consists of all the different other modules and the dependencies that they share.

adoxx-all-api-public

This module contains the code of the ADOxx ALL Public API. It is included in this repository since no public Maven repository is offered where this API is stored. The code is cloned from the repository [adoc] and not modified, other than the adoxx-ecore-converter module mentioned as parent.

converter-shared

This module contains interfaces that are used by a variety of modules in this project. To avoid duplicated code on any module, we created this shared library.

adoxx-to-ecore-converter

This module contains the logic for transforming an ADOxx metamodel to an Ecore metamodel. The class `ADOxx2EcoreTransformator` within this module contains the main procedure for the transformation.

The package `converter` contains semantically separated classes that create metamodel elements based on an input element. Where possible, those classes return a newly created element (e.g. a single attribute), which is added in the main-procedure class `ADOxx2EcoreTransformator`. The classes used in this package are:

- `ADOxx2EcoreAttributeConverter`
- `ADOxx2EcoreClassConverter`
- `ADOxx2EcoreEnumConverter`
- `ADOxx2EcoreInterrefConverter`
- `ADOxx2EcorePackageConverter`
- `ADOxx2EcoreRelationshipConverter`

The package `util` contains Util classes that are need on several other parts of the module, or that do not semantically fit inside the `converter` package, as they do not return target-metamodel elements. Those classes include:

- `ADOxx2EcoreClassFinder`
  (find classes based on a given name inside packages)
- `ADOxx2EcoreDuplicateNameResolver`
  (traverses through all metamodel elements and assigns new names to duplicates)
- `ADOxx2EcorePrinter`
  (Writes to a FileOutputStream and returns the location of the newly created file)

- `ADOxx2EcoreUtil`
  (contains several static util method like formatting a name to an EMF consistent name)

The package `metamodel.adoxx` contains the classes that are used by JAXB when unmarshalling an XML file and were auto generated with the tool JSONtoCSharp [jso].

ecore-to-adoxx-converter

This module contains the logic for transforming an Ecore metamodel to an ADOxx metamodel. The class `Ecore2ADOxxTransformator` within this module contains the main procedure for the transformation.

The package `converter` contains semantically separated classes that create metamodel elements based on an input element. Where possible, those classes return a newly created element (e.g., a single attribute), which is added in the main-procedure class `ADOxx2EcoreTransformator`. The following list shows the classes present in this package and provides an explanation where the meaning of a class name is not trivial:

- `Ecore2ADOxxAttributeConverter`
- `Ecore2ADOxxClassConverter`
- `Ecore2ADOxxCompositionConverter`
  (Creates and edits attributes and meta-information strings related to compositions on new or existing target metamodel elements)
- `Ecore2ADOxxMetaInformationConverter`
  (Assigns the meta-information strings used for multiple inheritance and compositions as well as the *ADOScripts* to the appropriate library attributes)
- `Ecore2ADOxxMultiInheritanceConverter`
  (Assigns non-yet-assigned attributes of a multi-inheritance relationships to classes and handles the procedure of incoming relationships on multi-inheritance classes)
- `Ecore2ADOxxRelationshipConverter`

The package `util` contains Util classes that are needed on several other parts of the module or that do not semantically fit inside the `converter` package, as they are not directly linked to creating and editing target metamodel elements. Those classes include:

- `Ecore2ADOxxPrinter`
  (Calls the ALL2ABL web service and writes the generated ALL and ABL file to a FileOutputStream)
- `Ecore2ADOxxUtil`
  (Contains several static util methods like finding all incoming relationships of an Ecore class)

adoxx-evaluator

This module contains the logic to create evaluation metrics for an ADOxx metamodel. It uses an `XmlAnalyzer` to analyze an ADOxx XML file and returns an instance of `EvaluationResult`, which is an interface defined in the module `converter-shared`.

ecore-evaluator

This module contains the logic to create evaluation metrics for an Ecore metamodel. It uses an `XmiAnalyzer` to analyze an Ecore XMI file and returns an instance of `EvaluationResult`, which is an interface defined in the module `converter-shared`.

executor

The module contains example main classes that a developer can use to trigger any module's results described above. We provided some classes to ease development and usage. These classes are listed and described here:

- `TransformDemo.java` (Sample execution procedure of both the ADOxx and EMF transformation programs with all available metamodels)
- `ADOxx2EmfEvaluation.java` (Sample execution procedure that evaluates ADOxx source metamodels and EMF target metamodels and collects the results in an Excel File)
- `Emf2ADOxxEvaluation.java` (Sample execution procedure that evaluates EMF source metamodels and ADOxx target metamodels and collects the results in an Excel File)

adoxx-ecore-rest-controller

This module contains a SpringBoot application that serves as the rest controller for the deployed web-app. It contains the following endpoints:

- @PostMapping("/adoxx-to-emf/upload")

  When navigating to this URL and providing an ADOxx metamodel file in the request body, the endpoints transforms this file by calling the transformation module internally and returns the generated file name upon creation.

- @GetMapping("/adoxx-to-emf/fileName")

  When navigating to this URL and specifying the file name via the URL, the endpoint returns the transformed metamodel file in the EMF xmi format (file name ending with `.ecore`) that was previously generated.

- @PostMapping("/emf-to-adoxx/upload")

  When navigating to this URL and providing an Ecore metamodel file in the request body, the endpoints transforms this file by calling the transformation module internally and returns the generated file name upon creation.

- @GetMapping("/emf-to-adoxx/fileName")

  When navigating to this URL and specifying the file name via the URL, the endpoint returns the transformed metamodel file in the ADOxx `.abl` format that was previously generated.

adoxx-ecore-converter-frontend

This module contains the Angular application that serves as the frontend for the deployed web-app. It contains two components, the `adoxx-to-ecore` component and the `ecore-to-adoxx` component. They possess the UI components and the frontend logic to interact with a specific endpoint of the module `adoxx-ecore-rest-controller`.

Prerequisites

The following tools are required in order to execute any of the described modules above:

Tool	Version
Java	$= 11$
Maven	$\geq 3.6.0$
node.js	≥ 16.0
npm	≥ 8.0

Execution

This section explains how to interact with the various executable of the code base.

Execution of distinct modules

To interact with distinct modules, they have to be built first. A user has to perform the following steps:

1. Navigate to the root folder of the `adoxx-ecore-converter` module
2. Execute `mvn clean install`
3. Wait for modules to be successfully built

Executing the modules can now be performed in two ways:

Either by creating an own Maven project and including the root module or any sub module:

```
<dependency>
   <groupId>at.ac.tuwien</groupId>
   <artifactId>adoxx-to-ecore-converter</artifactId>
   <version>1.0-SNAPSHOT</version>
</dependency>
```

Or use one of the existing execution files located in the module `executor`. Their execution can be triggered e.g., by an IDE (like IntelliJ), by navigating to the main method and selecting the execution option.

Execution of web application

To execute the web-application locally, the following steps have to be performed:

1. Execute the command `mvn clean install` on the root module `adoxx-ecore-converter` to build the whole application
2. Execute the controller application from an IDE or with the command `mvn spring-boot:run` inside the rest controller application folder
3. Execute `npm install` inside the folder of the module `adoxx-ecore-converter-frontend`.
4. Excecute the command `ng serve` to run the frontend application and then navigate to `http://localhost:4200`