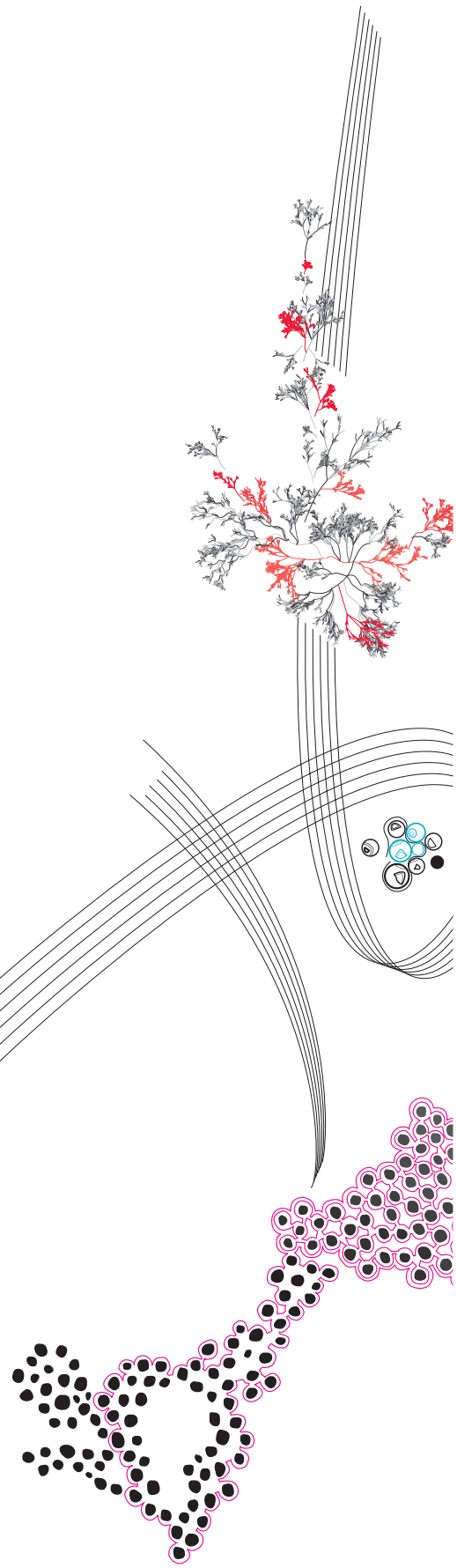


MSc Computer Science
Master Thesis

Ontology-Driven Software Development: Generating Java code from OntoUML

Guus Grievink

Committee: dr. Luís Ferreira Pires, Prof.dr. Arend Rensink, dr. João Luiz Rebelo Moreira

November, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

OntoUML is a modelling language intended for structural conceptual modelling using ontological theories from the Unified Foundational Ontology (UFO). Compared to plain UML, OntoUML is oriented towards the concept of ontologies and thus aims to capture more precise semantics about the modelled domain. For OntoUML to be of added value for the development of software applications, the ontological semantics should be respected by the developed code. When using an OntoUML model as a reference ontology, software developers might not be familiar with the underlying UFO theories of an OntoUML model, which might result in consistency issues between the created software and the underlying ontology.

The research provides an automated transformation from OntoUML to Java code. The goal has been to develop a transformation that preserves OntoUML semantics and that can be used in conjunction with existing OntoUML tools.

The transformation was developed within the Eclipse Modelling Framework (EMF) and is split up into three steps. The first step consists of parsing an OntoUML JSON file to an EMF-compatible Ecore model. The second step is to transform an OntoUML model into an implementation model, which is an intermediary representation in UML that lies close to the actual implementation of an application. The final step is the generation of Java code from the implementation model.

The implemented transformation is validated by executing it on publically available models from the OntoUML model catalogue. For a selection of 82 models, an automated validation effort tests whether the transformation does not yield errors and generates compilable Java code. In a manual validation effort, code generated from five models was inspected in more detail to check whether the generated code matched the expected patterns.

Among the contributions of this research are a transformation on a conceptual level of twelve OntoUML class stereotypes, an EMF Ecore metamodel for OntoUML that can be reused for other model transformation projects, and an implementation of this OntoUML-to-Java transformation. Furthermore, the research yielded insights about the current state of OntoUML tools which resulted in a list of recommendations.

Chapter 1

Introduction

This chapter describes the motivation and context of this research. Section 1.1 describes the role of ontologies and modelling in Software Engineering, introducing OntoUML as a means to better perform conceptual modelling. Section 1.2 describes the objectives of the research, namely to develop an OntoUML-to-Java code transformation. Section 1.3 describes the methodology on how to develop this transformation and how it will be validated. Finally, Section 1.4 describes the report structure.

1.1 Motivation

Modelling has always been a major activity in Software Engineering (SE). Models in SE can range from conceptual to technical models. A popular modelling language is the Unified Modeling Language (UML), which is defined by the Object Management Group[1] and provides a variety of model types for use in SE.

Ontologies provide a more formal means to perform conceptual modelling. In Computer Science, ontologies are explicit specifications of conceptualisations [11], which come in the form of machine-readable artefacts that are grounded in logical statements. Ontologies can play different roles in SE. One major distinction is between operational and reference ontologies [6][2].

Operational ontologies are lower-level specifications meant to be part of a system. Thus, the ontology is actively queried during the runtime of a system to retrieve some knowledge or information [6]. These operational ontologies are often defined in the Web Ontology Language (OWL) [24] and make use of other tools like SPARQL to query information [6]. In this field, several sources acknowledge that developers are not often familiar with the technologies behind ontologies, which is mostly the area of knowledge engineers. Both OBA [7] and JOINT [26] try to mitigate this by providing developer-oriented technologies to access ontologies. OBA does this by providing a REST API to query ontologies, where REST calls are mapped to SPARQL queries which retrieve information from OWL ontologies. Similarly, JOINT maps Java objects to an RDF4J[3] data storage application so that developers can use Java to query an OWL ontology stored in the RDF format.

In contrast to operational ontologies, reference ontologies are used as a reference for the implementation of software applications [6]. In this category, the role of ontologies is to add more quality criteria to conceptual models [28]. An example is OntoUML, which

[1] See https://www.omg.org/.

[2] While the terminology of reference and operational ontologies is used here, it is often also classified as ontologies used during the development time and runtime respectively [12, 23].

[3] See https://rdf4j.org/about/.

is defined as a lightweight extension to UML class diagrams to represent concepts from the Unified Foundational Ontology (UFO) [15]. UFO is an upper ontology which describes generic concepts expected to be used by more specific ontologies [12]. Hence, UFO supports representing things as objects, relations and time, which are domain-independent.

By grounding UML models in a foundational ontology, OntoUML models are of higher quality (i.e., they match reality better) compared to plain UML class diagrams [33]. Thus, we can claim that the software systems based on these models should also be of higher quality [28].

1.2 Research objective

This research aims to provide an automated transformation from OntoUML to Java, ensuring the underlying semantics are preserved as well as possible. When developing an application based on a reference ontology in OntoUML, it is desired that all added semantics of OntoUML are preserved in the application code, otherwise, one could question the added value of using OntoUML in the first place. Without an automated transformation, this relies on the developer being familiar with OntoUML concepts and their underlying ontological meaning. However, as mentioned before, there is often a knowledge gap between developers and knowledge engineers in this regard, so there is a risk of losing semantics when developing an application based on an OntoUML model by hand. By designing transformation rules that preserve OntoUML semantics one does not rely on the developers being familiar with the underlying UFO theories.

Furthermore, all currently available tools for OntoUML have the purpose of either editing OntoUML models or accessing/querying OntoUML models, making them more like operational ontologies. To the best of our knowledge, there is a lack of tools that support the development of applications based on OntoUML models (i.e., their use as reference ontologies). The availability of suitable tools can be considered a prerequisite to the adoption of OntoUML as a reference for implementing software applications.

For the OntoUML-to-Java transformation to be feasible, it should be of use to the OntoUML community. Therefore, this transformation should be compatible with other tools that allow the creation of OntoUML models. Furthermore, the validation of the implemented solution should include publicly available models as opposed to models solely created for testing purposes.

In this research, we aim to generate Java code. However, it is reasonable to expect that some people might want to generate code for other programming languages. Although the focus lies on the Java language, the structure of the transformation should allow the adaptation to generate code for other object-oriented programming languages.

1.3 Methodology

The methodology applied in this research is based on Design Science Research (DSR) [34]. Figure 1.1 shows the DSR steps and how they have been filled in for this research. The research presented in this report covers the process up until the demonstration step. User evaluation and possible feedback loops are subject of future research.

Design and develoment Before the OntoUML-to-Java transformation can be implemented, we should consider the design of the transformation. First, we should determine what OntoUML constructs to cover with our transformation. OntoUML consists of a variety of constructs, each of these vary in how frequently they are used in models and how

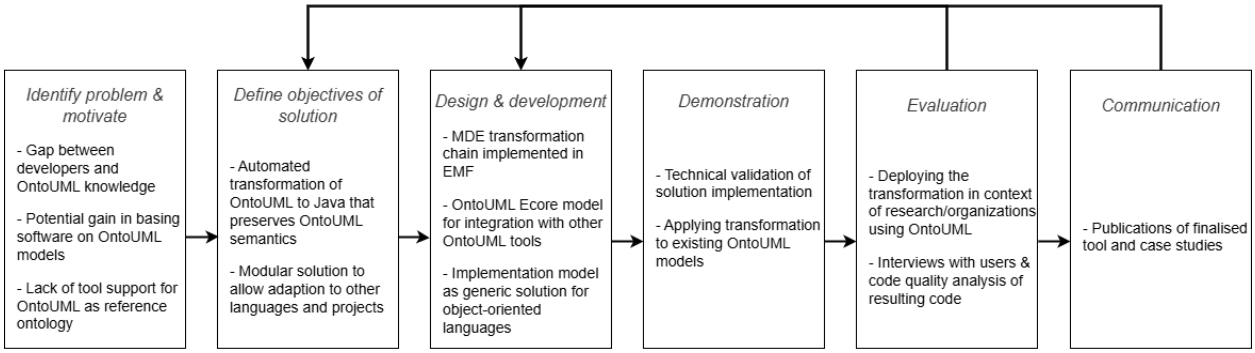


FIGURE 1.1: Design Science Research methodology filled in for this research. Based on [34, 31].

well they are documented. The selection of which constructs to support should take into account the wish to transform as many OntoUML models as possible as well as the availability of literature on the underlying UFO concepts. Regarding the latter, if we do not know the precise semantics, we can not preserve them.

Furthermore, to implement our transformation, we first need to design the transformation on a conceptual level. This includes considering the UFO theories behind OntoUML types and relating these to available constructs in the implementation model. Here, all the decisions are made on how to preserve the underlying semantics of OntoUML models in the generated code.

To implement the OntoUML-to-Java transformation, we have used Model-Driven Engineering (MDE) techniques. The transformation chain, which shows the different steps in the transformation, is illustrated in Figure 1.2. The technology used to implement the transformation is the Eclipse Modelling Framework (EMF)[4], which provides several tools for the application of MDE.

On the left-hand side in Figure 1.2, the starting point of the transformation can be seen, which is an OntoUML model. The first step is to transform the OntoUML model into a notation that is compatible with EMF. This model is then transformed into an implementation model, which represents an object model that is compatible with Java. The final step is to generate Java code from the implementation model.

To realize this transformation, the Atlas Transformation Language (ATL)[5] and Acceleo[6] have been used.

Demonstration As part of the demonstration, the developed transformation is tested on publicly available models from the OntoUML model catalogue. An automated validation is performed to see whether the models run through the transformation chain without resulting in errors and a check is made whether the generated Java code can be compiled.

Although this indicates the transformation being successfully executed, it does not guarantee that all OntoUML model elements generated the expected patterns in the Java code. Therefore, a manual validation will be performed on a small selection of OntoUML models.

[4]See https://eclipse.dev/modeling/emf/.
[5]See https://eclipse.dev/atl/.
[6]See https://eclipse.dev/acceleo/

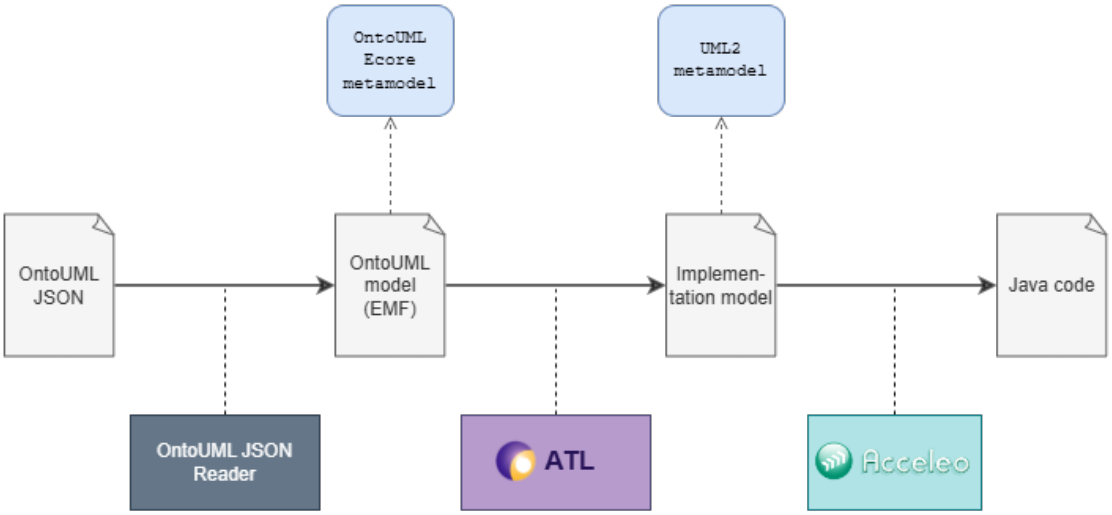


FIGURE 1.2: Overview of transformation from OntoUML to Java code

1.4 Report structure

This report is further structured as follows: Chapter 2 includes the background for this research. This covers ontologies in general as well as UFO and OntoUML specifically. Furthermore, it describes MDE and the EMF tools used to implement the transformation. Chapter 3 describes the transformation design, which includes the selection of OntoUML constructs to be covered, a description of the implementation model, and the transformation described on a conceptual level. Chapter 4 describes the integration of OntoUML within EMF, which includes the definition of an Ecore metamodel and a method to instantiate this model from OntoUML JSON files. Chapter 5 describes the technical implementation of the transformation of an OntoUML EMF model to Java, consisting of the implemented ATL transformation and Java code generation using Acceleo. Chapter 6 addresses the validation aspects of the implementation, including an automated process that verifies whether the generated Java code can be compiled as well as manual validation for a smaller selection of OntoUML models. Chapter 7 describes related work describing approaches that use OntoUML in a MDE context. Chapter 8 discusses remaining findings as well as insights on how the developed transformation can be extended/improved. We conclude with Chapter 9, which summarizes the contributions, lists future work, and lists recommendations for the OntoUML tools.

Chapter 2

Background

This chapter describes the technologies and literature sources that are used in this research. Section 2.1 discusses the concept of ontologies and its role in Computer Science. Within this section, OntoUML and the Unified Foundational Ontology (UFO) are highlighted, which both provide a means to incorporate ontological theories in conceptual modelling. Section 2.2 discusses Model-Driven Engineering (MDE) along with the Eclipse Modelling Framework (EMF), which provides open-source tools to work with MDE.

2.1 Ontologies

In Computer Science, the term ontology is adopted from the philosophical field of Ontology (singular with a capital 'O'). This is a millennia-old field that deals with the question of what there is [25]. This question is not trivial: different people can have different views of the world, either because of their viewpoints, personal beliefs or any other factor. Within Computer Science literature, such a view of the world is called a *conceptualization* [12][1]. For example, a farmer in Africa might form a conceptualization of African wildlife, which is based on his personal experiences. This probably differs from the conceptualization formed by a high-school student in the Netherlands reading about these animals in a biology textbook.

Concerning the definition of ontology (with a small 'o') in Computer Science, one of the older well-known definitions is from Gruber, who states that an ontology is *"an explicit specification of a conceptualization"*[11]. This definition can be criticized because of the vagueness of the terms 'explicit specification' and 'conceptualization' [28]. However, it still provides some handles to understand the nature of ontologies in Computer Science, namely, the definition implies that an ontology is not an idea that lives in someone's head but has to be an explicit specification, i.e., it has to be written down in some formal way. This means that an ontology in Computer Science is an artefact representing knowledge about some part of the world.

2.1.1 Ontology languages

Consider the scenario of an African wildlife expert wanting to define an ontology for this domain, as is discussed in [28]. Such a domain expert might be most comfortable capturing

[1] Although this is called a conceptualization in Computer Science, in Philosophy this is called an ontology (with a small 'o'). The different term in Computer Science was created to disambiguate between a conceptualization and the specification of that conceptualization which in Computer Science is called an ontology [12].

this knowledge in a written essay. However, natural language can be ambiguous, resulting in problems when several people interpret the text differently.

Another way to formally encode this knowledge would be in mathematical logical statements, such as first-order predicate logic. For the African wildlife domain, this would result in statements like:

$$\forall x(Lion(x) \Rightarrow \forall y(eats(x,y) \Rightarrow Herbivore(y)) \wedge \exists z(eats(x,z) \wedge Impala(z))) \quad (2.1)$$

This logical formula [28] states that lions only eat herbivores, and of all herbivores, lions are known to at least eat impalas.

Considering ontologies are artefacts in Computer Science, ontologies should be machine readable [22]. This is where ontology languages come into the picture, which provide a syntactical way to define ontologies. The language to define an ontology should respect the underlying mathematical logic of the ontology [28].

In SE, ontologies can be used in different ways. The most distinguishing is whether the used ontology serves as a reference or is an operational ontology [6], also referred to as ontologies used during development or the runtime of a system, respectively [12, 23]. Reference ontologies serve as a guide for the implementation of a software system and are hence used during development time. OntoUML was primarily devised to be used for this purpose [6]. Compared to reference ontologies, operational ontologies are lower-level specifications meant to be incorporated into a software system. In this fashion, they can be considered a component of the system which is queried for information. Because of this, they are subject to other requirements with respect to having desirable computational properties for the solution provided by the software system [6].

One example of the application of an operational ontology is the MOST workbench, which aims to be an extended Integrated Development Environment in which ontologies that describe the development process guide the user during the development of a system [30]. In contrast, an example of reference ontology can be found in [5], in which an ontology based on the Digital Platform Ontology is used to implement a minimal viable version of a digital platform.

2.1.2 Foundational Ontologies

Ideally, an ontology matches exactly the conceptualization to be represented, however, realistically this is not possible. The quality of an ontology can be defined by how well it matches its intention [28]. In other words, does the ontology properly cover what the modeller tried to represent?

Four categories of qualities of ontologies can be seen in Figure 2.1. In this figure, the part of the world that is modelled is denoted in pink (i.e., the conceptualization that the modeller tries to represent) whereas the green area indicates what is represented with the ontology.

With this representation, an ontology is considered 'good' when it covers slightly more than the part of the world that is being described while for a 'less good' ontology, this difference is larger. An ontology that does not describe all the things it should is considered 'bad'. Furthermore, the ontology is considered 'worse' if it is 'bad' and it also supports the modelling of things that should not be represented [28].

So, if one wants to make a 'good' ontology, one should ensure the ontology represents the intended subject as closely as possible. One approach is to make use of foundational

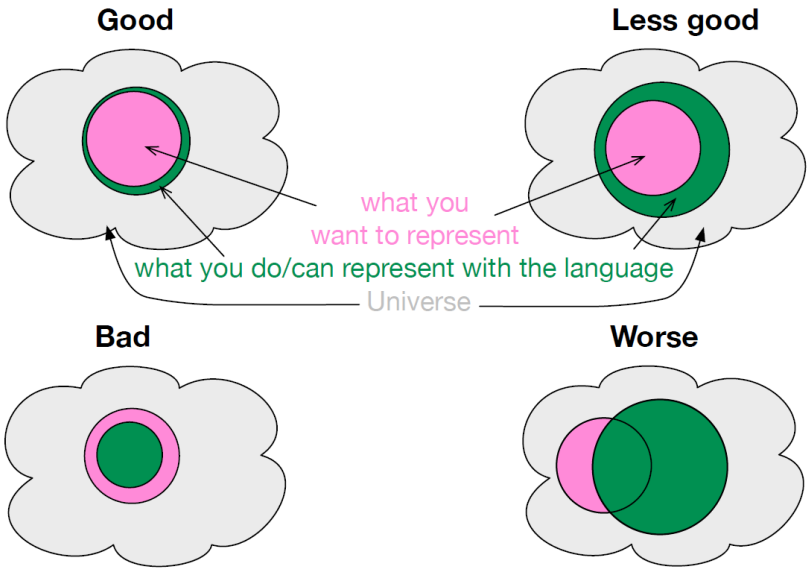


FIGURE 2.1: Good and bad ontologies based on the target domain and the things represented in the ontology. (Figure copied from [28])

ontologies [28]. Foundational ontologies, also known as upper or top-level ontologies, represent common concepts expected to be used by more specific ontologies [12]. Hence, they model things such as objects, relations, time, etc., which are domain-independent. Whereas more specific ontologies such as domain or task ontologies may refer to concepts expressed within foundational ontologies [12], foundational ontologies themselves are often solely founded in theories expressed in first-order logic or similar formalizations [24]. By using a foundational ontology, one reuses well-founded theories, which in turn potentially prevent unintended constructs and thus results in 'better' ontologies [28].

Many foundational ontologies such as BFO, DOLCE and UFO, have been developed over the last decades [1]. For this research, we concentrate on UFO as it is the basis for OntoUML.

2.1.3 OntoUML

OntoUML is a modelling language intended for structural conceptual modelling using ontological theories from UFO [21]. As the name suggests, OntoUML is a lightweight extension of the Unified Modelling Language (UML) [9], more specifically of UML class diagrams, realised by means of a UML Profile.

OntoUML has class and association stereotypes that allow the representation of UFO concepts. In UML diagrams, stereotypes are surrounded by guillemots (« and »). Figure 2.2 shows a small OntoUML model in which we see four different OntoUML types (*category*, *kind*, *phase*, and *subkind*). To work with OntoUML, several tools are available. The projects in which tools are developed are all included in https://github.com/OntoUML.

Visual Paradigm plugin Visual Paradigm (VP)[2] is a development tool suite which includes a modeller for UML. One of the OntoUML tools is a plugin for VP[3]. This plugin

[2]See https://www.visual-paradigm.com/.
[3]See https://github.com/OntoUML/ontouml-vp-plugin.

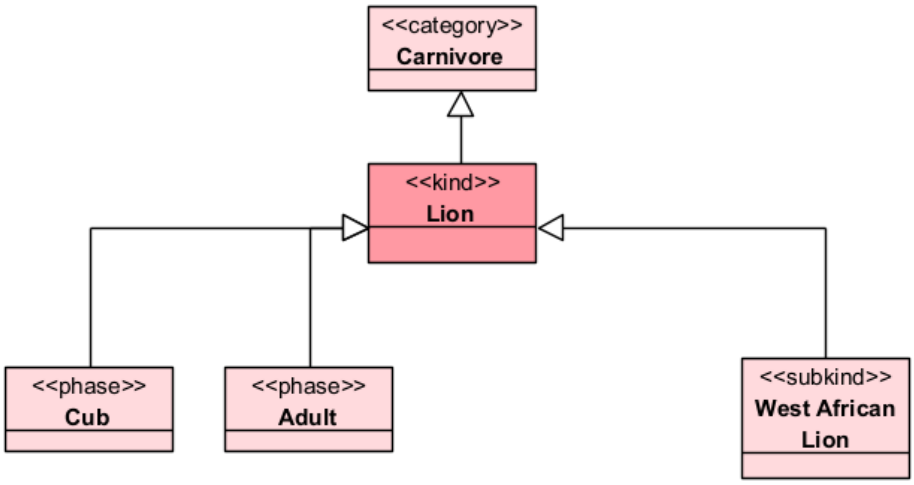


FIGURE 2.2: A simple ontology of lions made with the OntoUML plugin for Visual Paradigm.

installs the UML Profile which contains the UFO stereotypes. Next to this, it adds functionality to facilitate the editing of OntoUML models. This includes UI elements to add stereotypes, model verification, and smart colouring of classes. This smart colouring can also be seen in the different colours for kinds and the other OntoUML types in Figure 2.2, which was created using the VP plugin.

Implementation-independent metamodel The OntoUML metamodel[4] is an implementation-independent definition of the language. In this report, we will refer to it as the implementation-independent metamodel to avoid confusion with the OntoUML Ecore model described in Section 4.1 (which is also an OntoUML metamodel).

Even though OntoUML is an extension of UML, the platform-independent metamodel does not rely on UML. Thus, it is a standalone description of OntoUML models that can be used across the different OntoUML tools and support the interchange between them.

2.1.4 Unified Foundational Ontology

UFO finds its origins in the PhD work of Guizzardi [15]. It has primarily been developed for use in conceptual modelling and as such is tightly coupled to and co-developed with OntoUML [20]. UFO is a collection of micro-theories published across several papers. These micro-theories are split into several UFO parts: UFO-A (endurants), UFO-B (perdurants) and UFO-C (social/intentional concepts).

UFO separates concepts based on different existential properties. Figure 2.3 displays the taxonomy of UFO concepts. The major distinctions between these are based on four concepts: universals vs. individuals, endurants vs. perdurants, sortals vs. non-sortals, and rigid vs. anti-rigid. Furthermore, we will highlight the concept of identity-providers.

Universals vs. individuals

The distinction between universals and individuals is similar to classes and objects in object-oriented programming and traditional conceptual modelling. In UFO, universals

[4] See https://github.com/OntoUML/ontouml-metamodel?tab=readme-ov-file

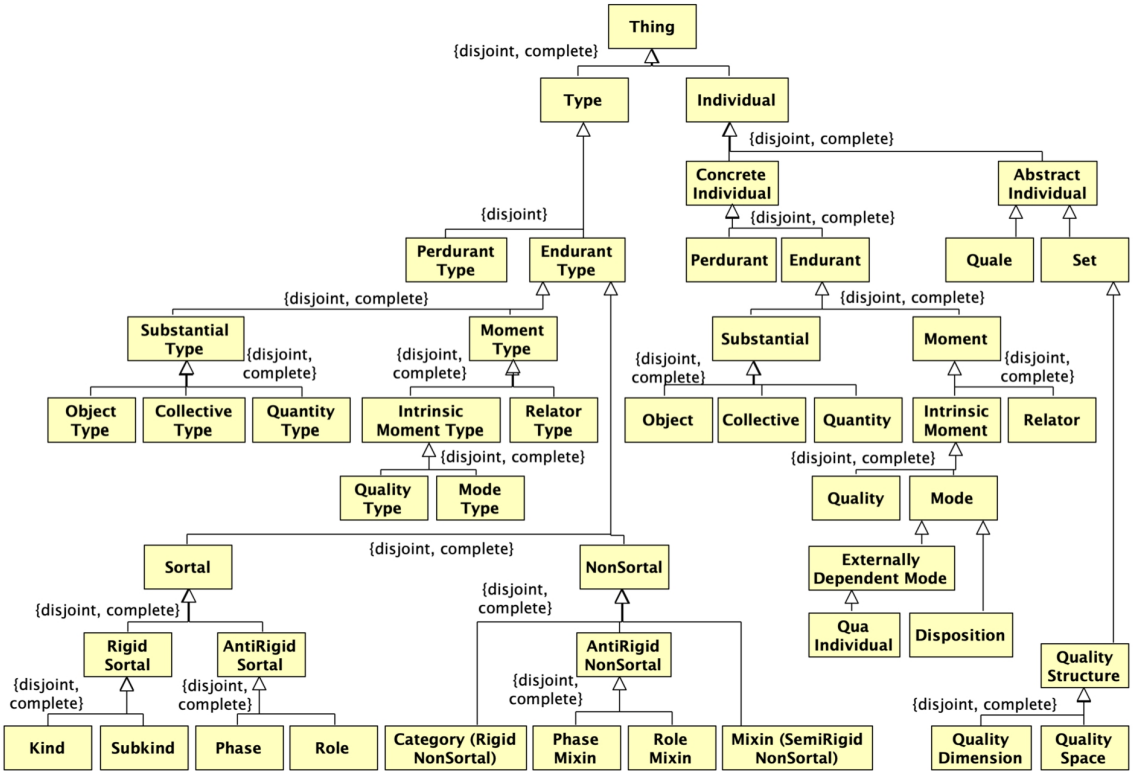


FIGURE 2.3: UFO taxonomy. (Figure copied from [20])

are "space-time independent pattern[s] of features, which can be realized in a number of different individuals" [16].

Another word for a universal is *'type'*, which is used in Figure 2.3. Based on Figure 2.3, it is apparent that this is the first distinguishing feature of UFO concepts.

Endurants vs. perdurants

An endurant is a type whose individuals exist in their entirety at some given point in time. A primary example of an endurant is a *'kind'* [20], such as the lion kind illustrated in Figure 2.2.

The opposite of endurants are perdurants, which unfold over time. The concepts relating to perdurants are all part of UFO-B, the primary concept being an *'event'*. Examples of events are the Second World War and the US 2024 elections. In this sense, perdurants themselves are processes that may change the state of the involved endurants. However, the perdurants themselves do not undergo change [20].

UFO-A, which describes endurants, is the most extensively described part of UFO. Nearly all concepts in the taxonomy on Figure 2.3 belong to this category.

Identity providers

Identity providers are sortal types that provide a principle of identity for their individuals. They are the OntoUML types *kind*, *collective*, *quantity*, *relator*, *mode*, and *quality* [20, 21]. In UFO terminology, these identity providers are all kinds, which is a distinct but related concept to the specific OntoUML type *kind*. This can be clarified by observing that in Figure 2.3, the *Endurant Type* is split into two distinct branches. The branch on the left

comprises the generalization set of *Substantial Type* and *Moment Type* and the branch on the right is the generalization set of *Sortal* and *NonSortal*. Both individual generalization sets are complete and disjoint, hence, a specific instance of a type has one classification based on the left branch and one based on the right branch. For example, there is a type which is both an *Object Type* and *Kind*. In OntoUML, this particular type is simplified into the stereotype *kind*. Table 2.1 relates the OntoUML stereotypes of identity providers to their corresponding classification from the UFO taxonomy illustrated in Figure 2.3.

In OntoUML, all these kinds can be specialized by the UFO types *subkind*, *phase*, or *role* [21]. For this research, we call these base sortals[5]. In the OntoUML VP plugin, different sorts of identity providers are assigned a specific colour: substantial types are coloured red, intrinsic-moments blue and relators green. The base sortals (that specialize these identity providers) are assigned a colour based on what identity provider they specialize. For example, in Figure 2.2, the base sortals all specialize a kind (substantial object type) and are given a lighter red colour.

A principle of identity for a certain type determines what properties uniquely identify an individual. As such, every individual must obey exactly one principle of identity [21]. This results in every type hierarchy being made up of possibly many base sortals with one identity provider at the top, which is also referred to as ultimate sortal [21, 15]. Thus, another definition of identity provider is a type that can be an ultimate sortal. In Figure 2.2, the ultimate sortal is the kind *Lion*. The OntoUML VP plugin visualizes this by giving this class a darker colour.

The notion of a single principle of identity is described in first-order logic in Equation 2.2. In this formula, $a :: b$ states that a is an individual which is an instance of the type b[6].

$$IdentityProvider(t) \wedge x :: t \implies \neg\exists y(IdentityProvider(y) \wedge x :: y \wedge y \neq t) \tag{2.2}$$

Sortals vs. non-sortals

Sortality is a further specialization of universal endurants. Sortals are types whose individuals always belong to the same identity provider. Equation 2.3 describes this definition of sortals in first-order logic.

[5]The term *base sortal* was derived from the implementation of the OntoUML VP plugin. However, neither this term nor an alternative seems to be used in the literature.

[6]Both [20] and [21] provide a formalization of UFO in modal logic. The equations used in this section serve merely as an illustration.

TABLE 2.1: OntoUML stereotypes of identity providers and their corresponding concepts within UFO.

OntoUML stereotype	**UFO Concepts**
kind	Object Type, Kind
collective	Collective Type, Kind
quantity	Quantity Type, Kind
quality	Quality Type, Kind
mode	Mode Type, Kind
relator	Relator Type, Kind

$$Sortal(t) \iff EndurantType(t) \wedge \exists i(IdentityProvider(i) \wedge \forall x(x :: t \implies x :: i)) \quad (2.3)$$

In Figure 2.2, the kind, subkind, and phases are all sortal types (as also becomes clear from Figure 2.3). For example, an individual lion of the subkind *West African Lion* will always be an instance of *Lion* (the ultimate sortal in this hierarchy).

A non-sortal is an endurant type which is not a sortal. In Figure 2.2, we see the *Carnivore* category as an example of a non-sortal. An instance of *Carnivore* could be a Lion. However, if we imagine this OntoUML sample to be a bit larger, for instance by adding a kind *Tiger* (which is also a carnivore), this is not necessarily true anymore. In that case, an instance of *Carnivore* could also be a tiger, which provides a different principle of identity as the *Tiger* would be an ultimate sortal in another type hierarchy. This means that non-sortals never provide a principle of identity for their instances. Thus, by definition, non-sortals are abstract in OntoUML.

Rigid vs. anti-rigid

Rigidity is a distinction that applies to endurant types and both sortals and non-sortals. Rigid types are defined as the endurant types whose instances necessarily always instantiate this type [20, 21]. This definition is captured in Equation 2.4 which makes use of the modal operators for possibility and necessity ($\Diamond$ and $\Box$ respectively).

$$Rigid(t) \iff EndurantType(t) \wedge \forall x(\Diamond(x :: t) \implies \Box(x :: t)) \quad (2.4)$$

Anti-rigid types thus allow their instances to cease being an instance of that type. The two anti-rigid types are phases and roles (along with their respective non-sortal mixin types).

In Figure 2.2, we see two phases for the *Lion* kind: *Cub* and *Adult*. An instance of *Cub* is not necessarily always a cub. After ageing, a cub can become an adult, in which case it is no longer an instance of *Cub*. In contrast, subkinds are rigid, so for example, an instance of *West African Lion* would always be a West African lion, even if it were to be moved to a sanctuary or zoo in Europe.

2.2 Model-Driven Engineering

MDE is an approach in which models are the primary artefacts to be delivered as opposed to things such as code or documentation [2]. Other model-related approaches related to MDE are Model-Based Engineering (MBE), Model-Driven Development (MDD), and Model-Driven Architecture (MDA), each with its specific nuances. This collection of related but slightly different approaches is called the MD* jungle by [2]. Figure 2.4 shows how the MD* approaches are related to each other.

MBE can be considered the more general form among the different MD* approaches. In MBE, models play an important role in the different engineering processes but, as opposed to MDE, are not considered the primary artefacts. MDD is a more specific approach to MDE. Whereas MDE includes models for several engineering processes, such as reverse engineering and software evolution, MDD only uses models in development tasks [2].

Finally, MDA is a specific approach of MDD defined by the Object Management Group [8]. MDA proposes a set of principles for applying MDD that aid in deriving

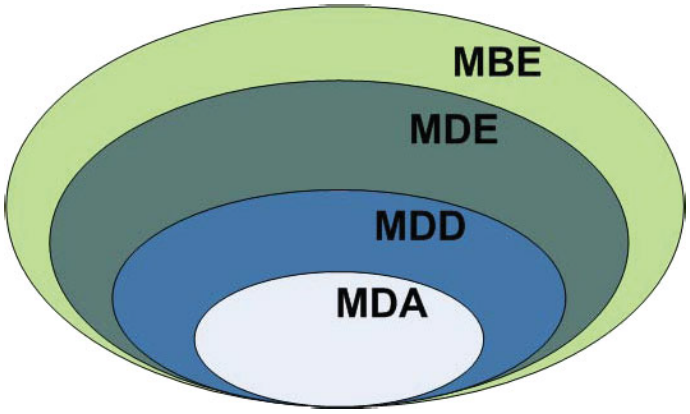


FIGURE 2.4: Relationship between different MD* acronyms. (Figure copied from [2])

value from models and help deal with complex systems. These principles describe the different models that can be defined for a system, such as models from various viewpoints and on different abstraction levels, as well as how these models interact and can be executed. Besides these guiding principles, MDA describes the usage of OMG-specific industry standards to define and work with models, which in the end defines what models are considered 'MDA Models' [8]

The process of working with MDE can be summarised in two challenges. The first is the question of what models actually are and how to define them. The second is how to go from a model to an executable artefact, such as software code. In the following sections, these questions are answered by describing the concepts of metamodels and model transformations. Furthermore, we discuss some Eclipse Modelling Framework (EMF)[7] tools related to these concepts that can be used to implement MDE solutions.

2.2.1 Models and metamodels

Models describe some part of the world [2, 30]. The subject of a model can be many things, including software systems [36]. Models provide an abstraction of their subjects, and they thus provide a simpler view of reality [30]. The model's goal is to capture the relevant properties of the real world such that it aids in accomplishing a specific task [2].

The degree of abstraction of models can be extended to multiple levels. Whereas models describe the real world, metamodels are models that describe (and hence are an abstraction of) other models. In MDE, a metamodel describes the modelling language in which a model is defined. Metametamodels, in turn, describe metamodels [22, 30, 2] and thus can be seen as the description of a modelling language that is used to define metamodels. This results in a metamodel hierarchy, which can be visualised as a pyramid with at the bottom the real world and at the top the metametamodel. Each of the levels can be filled in for a specific modelling scenario. Figure 2.5 includes the different metamodel levels that appear in the specific modelling technologies of MDA by OMG [8]. The real-world subject is located at the M0 level. M1 contains the model which is a UML class diagram in this case. The metamodel for a class diagram is UML and is located at the M2 level. The highest level is M3 and contains the Meta Object Facility (MOF) [10], which is another OMG standard and is the modelling language for UML. MOF is defined by itself, i.e., MOF acts as its own metamodel. Therefore, this is the highest metamodelling level and hence there is no

[7] See https://eclipse.dev/modeling/emf/.

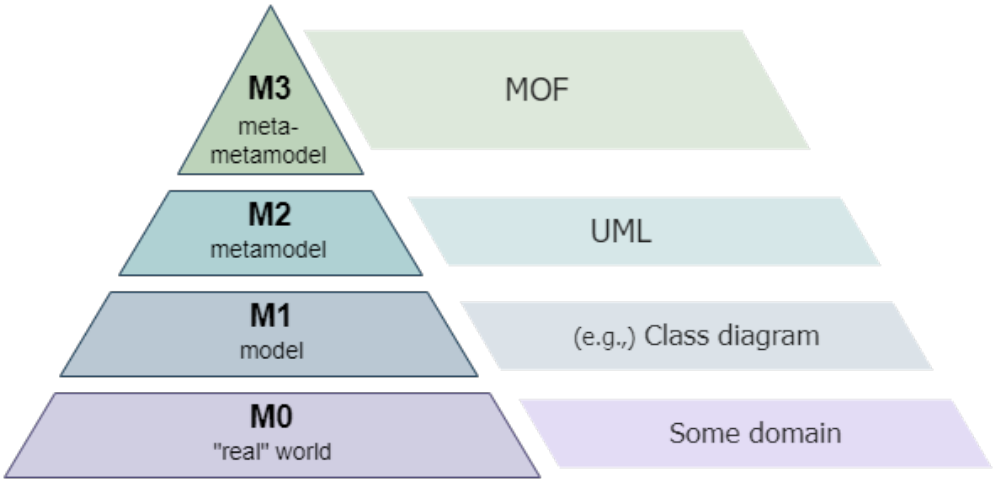


FIGURE 2.5: Metamodel pyramid for the UML language.

need to define an infinite number of levels [30, 2].

The metamodel pyramid in Figure 2.5 is based on the MDA-specific technologies MOF and UML. However, EMF works with another metametamodel. Figure 2.6 visualizes an example of a car that is modelled with a custom car metamodel defined in EMF. Starting at the M3 level is Ecore, which is the EMF equivalent of MOF. In Figure 2.6, a small excerpt of Ecore is represented that only contains *Class* and *Property*. On the M2 level, an instance of an Ecore model is presented. This Ecore model is defined in EMF and can serve as a custom modelling language. In this simple example, we have created a *Car* class which has two properties: one representing the brand and one indicating whether it has a radio. EMF also has tools to create instances of these custom metamodels. In Figure 2.6 at the M1 level, a simple XMI editor has been used to create an instance of a car. In this XMI editor, model elements are listed in a tree view (not visible in Figure 2.6 as there is only one model element) where the property of each model element can be altered. In this specific instance, the property *brand* has been set to *Saab*, corresponding to the car at the M0 level and the property *Contains Radio* is set to *true*.

2.2.2 Model transformations

Model transformations provide a means to generate new models from existing models with possibly different purposes as the goal. There are two general categories of model transformations: Model-to-Model (M2M), which takes a model as input and outputs another model, and Model-to-Text (M2T), which takes a model as input and outputs text, which can be, for example, code for some programming language [2].

Model-to-model transformations

In general, M2M transformations take any number of models as input and yield any number of other models as output. For most use cases, however, a transformation has only one input and one output model (also called source and target models, respectively). A common example of a model transformation is visualised in Figure 2.7. In this case, the transformation takes in *Model A* and outputs *Model B*. Both models are instances of their respective metamodels. A transformation is called exogenous if both input and output

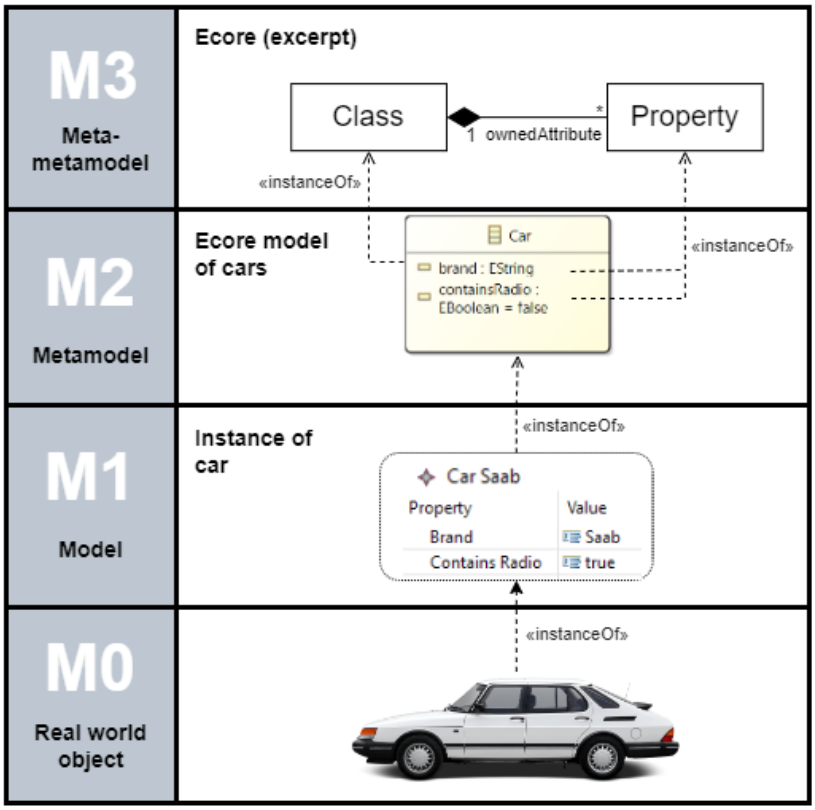


FIGURE 2.6: Example of different metamodel levels in the context of EMF.

models are an instance of the different metamodels, as is the case in Figure 2.7 assuming $MetamodelA \neq MetamodelB$. Alternatively, a transformation can be endogenous when both models adhere to the same metamodel. The latter can be useful for model refactoring, in which a model is restructured to improve its quality [2].

A transformation definition defines mappings to relate elements of the input model to elements in the output model that should be generated. To create a transformation definition, one can either use a Domain-Specific Language (DSL) for model transformations or a Generic Programming Language (GPL, e.g., Java or Python). A transformation definition defines mapping on the level of the metamodels, which ensures that the transformation can be applied to all possible instances of the source metamodel [2].

Atlas Transformation Language ATL[8] is a domain-specific language for model transformations. The language mixes declarative and imperative programming, which gives the freedom to solve varying transformation problems in different ways [27]. The transformation definition of ATL is itself a model conforming to the metamodel of the ATL language [30]. In the EMF implementation of ATL, all metamodels are an instance of Ecore, which is also visualised in Figure 2.7.

Listing 2.1 shows an ATL transformation that transforms a car model (from the metamodel in Figure 2.6) into a UML model. On line 5, we see the definition of the metamodels. UML is the target metamodel, and CAR is the source metamodel (corresponding to Metamodel B and Metamodel A illustrated in Figure 2.7, respectively). The main constructs of ATL are rules. Matched rules are declarative constructs that relate a source pattern

[8]See https://eclipse.dev/atl/

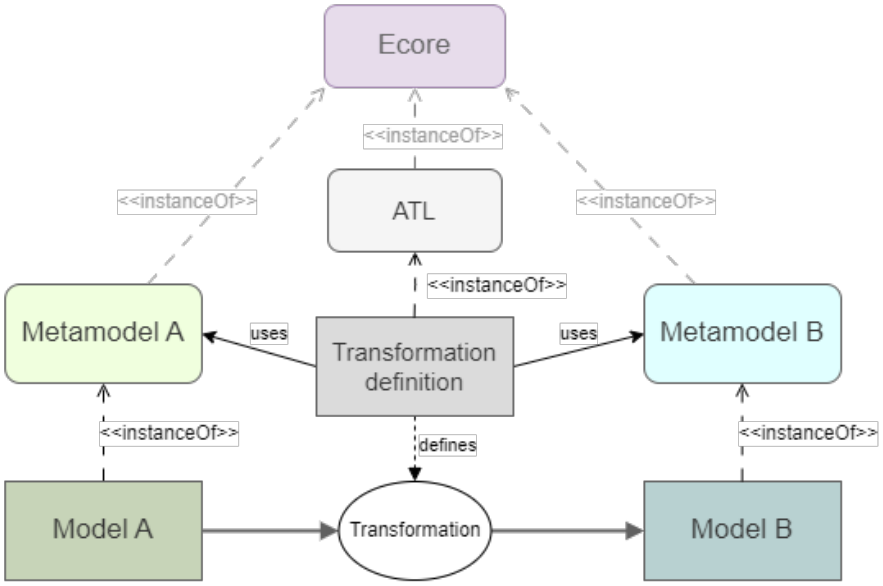


FIGURE 2.7: Pattern of a model-to-model transformation defined with the Atlas Transformation Language (ATL).

to a target pattern [27]. The rule *Carr2Class* in Listing 2.1 is an example of a matched rule. In this example, the source pattern (in the *from* clause) matches any instance of *Car* from the source model. For each of these cars, a UML Class is generated (defined in the *to* clause). In this case, the name of the UML Class is set to the *brand* property from the car.

Besides declarative matched rules, ATL contains two imperative constructs: called rules and action blocks. Opposed to matched rules that are executed automatically by ATL based on elements in the source model, called rules are executed either at the start or end of a transformation or are called in an action block. Action blocks are code pieces containing imperative statements, such as conditions, loops, and assignments [27]. They appear in the *do* clause of rules, as can be seen in Listing 2.1. This action block checks whether the car contains a radio, and if so, calls the called rule *createRadio()*. This called rule, recognisable by the parentheses (which may contain arguments), simply generates a UML Class with the name *Radio*.

The example illustrated in Listing 2.1 could be extended by, for instance, generating a UML Association that connects the car with the radio. Besides the constructs in this example, ATL contains other features that can be used, such as helper functions and lazy rules. Besides these ATL-specific constructs, OCL queries can be used to navigate through and select source model elements[9].

Model-to-text transformations

M2T transformations often provide the final step in transformation chains by generating text, which is usually code in some programming language. In an ideal world, one can generate a complete software system or application from a model. However, in practice, this is often not possible. Partial code generation can either mean that only some components

[9]The language documentation of ATL can be found on https://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language.

```
-- @path Car=/package.cars/model/cars.ecore
-- @nsURI UML=http://www.eclipse.org/uml2/5.0.0/UML

module carTransformation;
create OUT : UML from IN : CAR;

-- Matched rule that is executed by ATL for every Car
rule Car2Class {
  from
    c : CAR!Car
  to
    uc : UML!Class (
      name <- c.brand
    )
  do {
    if (c.containsRadio) {
      thisModule.createRadio();
    }
  }

}

-- Called rule that is only executed if called by code
rule createRadio() {
  to
    rc : UML!Class (
      name <- 'Radio'
    )
}
```

LISTING 2.1: Example of a ATL transformation using a matched and called rule.

of an application are generated, or that the generated code needs to be implemented further by a developer [2].

Just as with M2M, M2T transformations can be implemented in GPLs or DSLs. These DSLs for M2T transformations are also called code generators, most of which are based on template engines [2]. Figure 2.8 shows how a template engine uses a template and input model to generate text. The template functions as a blueprint for the to-be-generated code. Next to static code (which appears as is in the generated artefacts), it contains meta-markers that serve as placeholders for text that is generated based on information from the model. The template engine then is executed for a specific input model, fills in the meta-markers, and outputs text over possibly multiple files [2].

Acceleo Acceleo[10] is a mature M2T tool available in EMF. Listing 2.2 shows a minimal Acceleo file that generates a Java class from a UML model (for instance, the UML model generated by ATL from the example in Listing 2.1).

At the top of the file, the metamodel is defined. Similar to M2M transformations, an Acceleo template is defined on the metamodel level to work for all model instances. Furthermore, the EMF Acceleo plugin provides utilities such as code completion for these metamodel elements.

All the sections in square brackets ('[' and ']'), are part of the Acceleo language. The *template* section indicates a section of the to-be-generated text, in this case specifically for

[10]See https://eclipse.dev/acceleo/.

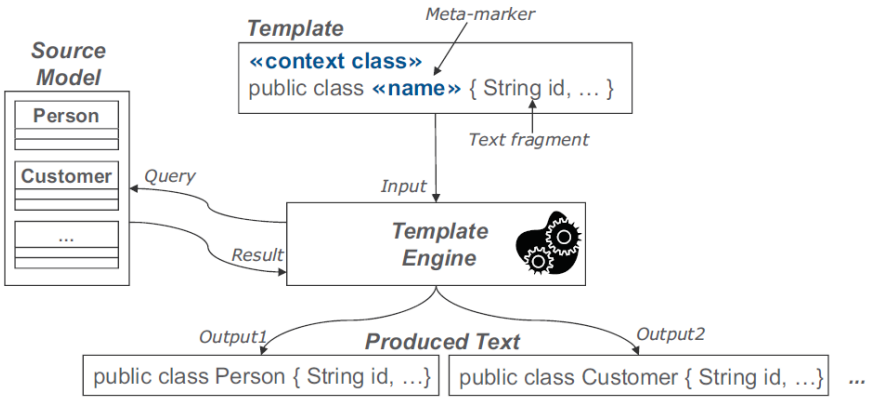


FIGURE 2.8: Relation between a template engine, the template, model, and output files. (Figure copied from [2])

```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/uml2/5.0.0/UML')]

[template public generateElement(aClass : Class)]
[comment @main/]
[file ('model/' + aClass.name + '.java', false, 'UTF-8')]
package model;

public class [aClass.name.toUpperFirst()/] {

  public [aClass.name.toUpperFirst()/]() {
    // TODO implement constructor
  }
}

[/file]
[/template]
```

LISTING 2.2: A simple Acceleo template that generates a Java class for UML Class elements.

the UML Class element. The *file* section indicates the result should be in its own file, with the filename being based on the Class name (a model property) and ending with *'.java'* (as is required for Java files).

The next pieces of text are part of the static code (displayed in black). These are Java language strings that are not interpreted by Acceleo. Within the static code, some meta-markers are defined, again enclosed in square brackets. In both meta-markers, the name of the UML Class of the input model is inserted. The first letter of this name is capitalised using the Acceleo *toUpperFirst* function, which ensures Java code style conventions. Next to this function, Acceleo contains many similar string functions that facilitate the creation of templates [2].

Chapter 3

Transformation design

This chapter describes the prerequisites for implementing our transformation from OntoUML to an implementation model. Section 3.1 describes the selection of OntoUML stereotypes that are covered by our transformation. This selection is based on OntoUML literature as well as an analysis of the OntoUML model catalogue, which contains a variety of ontologies created by the community.

Section 3.2 describes our implementation model, which is based on the UML Class diagram. In this section, we discuss the restrictions we imposed on these UML models and our choice of using specific UML attributes.

Section 3.3 describes our OntoUML-to-implementation-model transformation on a conceptual level. For each of the OntoUML types, a fragment of OntoUML is given along with the implementation model pattern to be generated.

3.1 OntoUML stereotypes to be covered

Since there is no single official specification of OntoUML yet, at the moment of writing, there is no clear definition of the valid OntoUML stereotypes. Varying options are described in the literature (such as [21] or [20]), in the platform-independent metamodel (described in Section 2.1.3), or on the online OntoUML documentation[1].

Our goal has been to design a transformation that both preserves the underlying UFO semantics and is of use to the community. The former requires that the to-be-transformed OntoUML types are well-described in the literature. Nonetheless, if we want to preserve the semantics, we should know what this semantics is. Regarding the latter, one could see what stereotypes are used most often. To answer this question, we checked the OntoUML Model Catalogue[2], which contains 168 OntoUML ontologies covering a wide range of domains.

In our analysis, we only consider the class stereotypes. The relation stereotypes in OntoUML are subordinate to the class stereotypes, i.e., relation stereotypes are specifically used for certain class stereotypes. Therefore, we covered these relation stereotypes together with their related class stereotypes.

In this section, we will first check the model catalogue to gather statistics on the most commonly used stereotypes. Together with information regarding the availability of literature on OntoUML types, this inspired our final selection of OntoUML types to be covered, which is given at the end of this section.

[1] See https://ontouml.readthedocs.io/en/latest/index.html.
[2] See https://github.com/OntoUML/ontouml-models.

3.1.1 Scraping the OntoUML Model Catalogue

The model catalogue contains 168 OntoUML ontologies. For each project, several artefacts are included, such as the Visual Paradigm project file, exported images of the ontology, the ontology itself (in both the JSON and Turtle format), and metadata.

For each ontology, we use the JSON representation to find the frequencies of stereotypes for both classes and relations. To achieve this, a Python script was used[3]. With this information, we can determine how often each stereotype is used in each OntoUML model.

3.1.2 Stereotype frequency

With the data from the model catalogue, we can calculate two different frequency metrics: the number of projects in which a certain stereotype occurs and the total cumulative occurrence of a stereotype across all models.

The complete results list can be found in Appendix A. In Table 3.1a, the 10 stereotypes that appear in most projects are listed. From this table, it follows that *kind* is the most common stereotype, occurring in 136 out of the 168 ontologies.

In Table 3.1a, at place 10, we see the stereotype *event* which belongs to UFO-B. According to [21], *"OntoUML, as all structural conceptual modelling languages, is meant to represent type-level structures whose instances are endurants [...]"*. As such, one could argue that UFO-A types are more fundamental to OntoUML than UFO-B or UFO-C, especially in the context of generating code, as the generated code represents the structure of entities within a domain. Therefore, in our selection of what stereotypes to cover, we limit ourselves to UFO-A types. In Table 3.1b, a top 10 is provided that is filtered for valid stereotypes belonging to UFO-A.

Total cumulative occurrence of stereotypes

Besides looking at the number of projects a stereotype occurs in, one can also calculate the total cumulative occurrence of a stereotype, which also takes into account how many times a stereotype occurs in a single project. Table 3.2a displays the top 10 stereotypes according to this metric. In contrast to the number of models a stereotype occurs in, not *kind*, but *subkind* is the most frequent stereotype; 2215 classes in the model catalogue have this stereotype.

When looking at the rest of the top stereotypes, *Goal* and *Null* spring to attention, which, according to the OntoUML platform-independent metamodel, are not valid OntoUML types. As these are not valid OntoUML stereotypes, it also does not seem useful to design a transformation for them. More non-valid stereotypes occur outside the top 10. In the results in Appendix A, a column indicates whether the stereotype is valid according to the OntoUML metamodel. Table 3.2b displays the top 10 most frequent stereotypes when these are filtered for valid UFO-A stereotypes[4]. Worth noting is that this top 10 comprises the same 10 stereotypes that occur in Table 3.1b, albeit in a different order.

The occurrences of invalid stereotypes give rise to some other questions. For example, one might be surprised to see the high occurrence value of the invalid *Goal* stereotype in Table 3.2a. However, when looking further, there is only one ontology containing this stereotype, i.e., one ontology in the model catalogue contains 951 classes with *Goal* as the stereotype. This is problematic when measuring commonality, so additionally, one would

[3]This Python script can be found at https://github.com/GuusVink/GeneratingJavaFromOntoUML-auxiliary

[4]This table filters both for valid stereotypes according to the platform-independent metamodel as well as stereotypes originating from UFO-A.

Table 3.1: Number of occurrences of stereotypes in OntoUML models from the model catalogue.

(a) Overall top 10 according to occurrence in ontologies.

Stereotype	Occurs in # ontologies
kind	136
relator	125
role	116
subkind	116
category	96
roleMixin	65
mode	63
collective	58
phase	56
event	49

(b) UFO-A top 10 stereotypes according occurrence in ontologies.

Stereotype	Occurs in # ontologies
kind	136
relator	125
role	116
subkind	116
category	96
roleMixin	65
mode	63
collective	58
phase	56
quality	42

Table 3.2: Top 10 class stereotypes in order of their total cumulative occurrence.

(a) From the model catalogue.

Stereotype	Total occurrence
subkind	2215
role	2180
kind	1843
relator	1405
Goal	951
null	786
category	611
roleMixin	571
mode	552
event	447

(b) Filtered for valid stereotypes belonging to UFO-A.

Stereotype	Total occurrence
subkind	2215
role	2180
kind	1843
relator	1405
category	611
roleMixin	571
mode	552
phase	383
quality	215
collective	184

like to determine in how many ontologies a specific stereotype occurs. In that case, a stereotype is considered common if it is contained in many ontologies even if, for example, it only occurs once in each of them.

Analyzing the number of models that could be transformed by a selection of stereotypes

In the selection of OntoUML stereotypes that are covered by our transformation, we want to maximize the number of OntoUML models that can be transformed. We say an OntoUML model can be transformed if it contains no class stereotypes that are not covered by our transformation. Mathematically, this can be expressed as:

$$supported(x) \iff class\_stereotypes(x) \subseteq supported\_stereotypes \tag{3.1}$$

in which x is an OntoUML model, *supported* is the predicate indicating whether a model can be transformed by the transformation, *class_ stereotypes* retrieves the set of class stereo-

types that occur in a model, and *supported_ stereotypes* is the set of class stereotypes that are covered by the transformation.

For a given set of supported stereotypes, the number of OntoUML models from the catalogue that would be supported by such a transformation can be calculated. When taking the top 10 stereotypes according to occurrence from Table 3.1b and Table 3.2b, 60 models from the model catalogue could be transformed.

3.1.3 Final selection of stereotypes

To make a selection of stereotypes to cover, we start with the eleven stereotypes defined in [21]. This is because it describes a new metamodel for OntoUML (termed OntoUML2) and thus seems most aligned with the future direction of OntoUML. Furthermore, the UFO semantics of all these stereotypes are well-defined.

There is a high level of correspondence between the types presented in [21] and the top 10 UFO-A stereotypes found in the model catalogue (which appear in Table 3.1a): nine out of these ten stereotypes match. The only stereotype from the top 10 which is not mentioned in [21] is *mode*. However, this stereotype is described in other authoritative literature, such as [18] and [20]. Therefore, we included this stereotype in order to extend the number of OntoUML models that can be transformed with our transformation. This leads to the selection of the following twelve stereotypes:

- kind
- subkind
- role
- phase
- category
- roleMixin
- phaseMixin
- mixin
- relator
- quality
- mode
- collective

When also including the *enumeration* and *datatype* stereotypes, which are supported by OntoUML but not specific to UFO theories, a transformation for this selection of stereotypes would cover 82 of the models from the OntoUML model catalogue.

3.2 Implementation model

Our implementation model lies close to the actual implementation of a Java program, such that the generation of code from this implementation model can be considered trivial. By transforming OntoUML into an intermediary form, we provide a more general solution as opposed to directly generating code. This facilitates the possible generation of code for multiple programming languages that all utilize the same OntoUML-to-implementation-model transformation.

As a basis for the implementation model, we use the class diagram concepts of UML. However, for the purpose of generating code, we apply a restriction on these class diagram concepts. UML allows for multiple inheritance while this is not possible to implement in Java. Thus, our main restriction is that the implementation model should not contain multiple inheritance. This means that a subtype may not inherit from multiple supertypes through a generalization. However, Java allows the implementation of multiple interfaces as well as interfaces extending multiple other interfaces. Therefore, no restrictions apply to the *interface realization* relation as well as generalizations for interfaces.

3.2.1 Final properties

In addition, we introduce a notice of usage of the UML model. In our implementation model, we wish to indicate that some properties of classes are *final* (following the Java terminology), which means that a reference to another object may not be changed. Some constructs in OntoUML imply these restrictions, such as the relation between a kind and a role (the role of a kind may change, yet a role always points to the same associated kind). This is further elaborated, along with an example, in Section 3.3.2.

We want this restriction to end up in the Java code as it matches the semantics of the underlying ontology. Firstly it restricts the Java runtime execution and secondly, it communicates some of the OntoUML semantics to the developer, without the developer being necessarily familiar with OntoUML.

For this information to be transferred into code, it should also be captured in the implementation model. This *final* restriction applies to specific ends of an association. Since the UML Property model element encompasses both basic properties of classes as well as properties that are represented by association ends, we considered two meta attributes of properties, namely *isReadOnly* and *isLeaf*.

isReadOnly The *isReadOnly* attribute belongs to structural features and, according to the UML specification, indicates that its value may not be modified [9]. Conversely, when *isReadOnly* is false, the value may be modified. For our purposes, the *isReadOnly* attribute seems too restrictive because, in our scenario, the instance of an object which is pointed to may be changed, i.e., functions of the object may be called such that its state is altered, as long as it remains the same object. The principle of identity of this object is assumed to be the reference Java keeps track of.

isLeaf The *isLeaf* attribute of a property is inherited from the UML *RedefinableElement*. It indicates that the element may not be redefined. In the context of classes, this is relatively straightforward, namely it states that no specializations of a leaf class should exist. Considering what it means for a property to be redefined, this seems to align with what we want, namely, the reference may not be altered, but the state of the value may be altered by the bearer of the property.

Java uses the *final* keyword to indicate classes that may not be extended. Thus, at least from the perspective of Java, there are some similarities in functionality between *final* attributes and the *isLeaf* attribute.

Furthermore, the Obeo UML-Java code generation project (further described in Section 5.4) also uses the *isLeaf* attribute for *final* attributes. Even more so, that transformation only generates get methods for leaf attributes and no set methods. Therefore, in our transformation, we made use of the *isLeaf* attribute to specify properties that should be marked *final*.

3.2.2 Visualisation of isLeaf attribute

In UML, the *isLeaf* attribute is not visualised in class diagrams. For illustration purposes, when visualising an implementation model, we added an icon to association ends in case the *isLeaf* attribute is set to true.

This icon can be seen alongside two classes and an association in Figure 3.1. The icon consists of a lock with the outline of a leaf within. In this example, *classA* has a property *propB* of type *ClassB* that is marked as *isLeaf*. Thus, this reference may not be updated

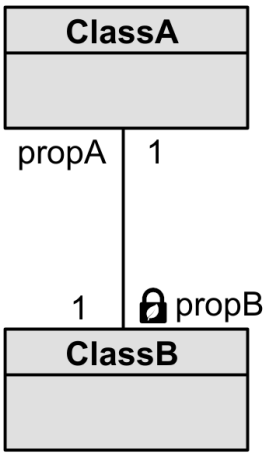


FIGURE 3.1: Visualisation of association end properties marked as final with the *isLeaf* attribute.

and will always refer to the same instance of *ClassB*. We refer to these association ends as being marked *final*, corresponding to the Java language construct of this property.

3.3 Transformation design of OntoUML types

This section describes our OntoUML-to-implementation-model transformation on a conceptual level. A transformation rule is provided for each OntoUML stereotype, consisting of a source model (a snippet of OntoUML containing the respective type), a target model (desired output of the transformation), and a rationale based on the underlying UFO concepts. For the source models, examples of typical uses of the OntoUML stereotypes are used according to both the online OntoUML documentation as well as literature describing OntoUML (such as [21, 20]). The relation stereotypes related to the class stereotype are also covered.

The transformations are visualised using the Visual Paradigm tool with the OntoUML plugin. The source models can be recognised by the OntoUML stereotypes and the different colours added by the OntoUML plugin. The target models are plain class diagrams coloured in grey. Each source model represents real concepts for illustration purposes. However, the provided transformation rules apply to all similar OntoUML constructs, independent of the name chosen for the type. Next to this, whenever possible, the source models represent certain OntoUML patterns. These patterns are generic structures of OntoUML that are considered best practice and are advised to be used whenever applicable [19].

Furthermore, a model element in the target model derives its name from the source model element. In case the name would be exactly the same, an apostrophe is added. This is done because Visual Paradigm requires unique names for different elements. The apostrophe is omitted in the implemented transformation and in the generated code.

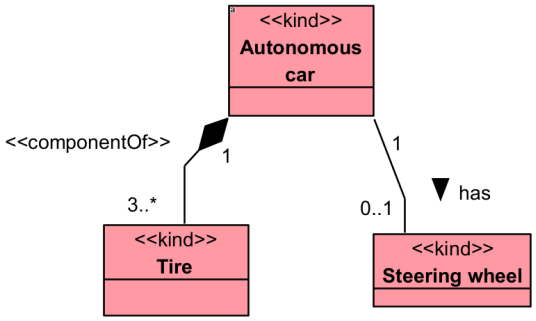
We discuss the proposed transformation of the different OntoUML stereotypes based on the category of UFO concepts they belong to. Section 3.3.1 covers UFO substantials (kinds and collectives), Section 3.3.2 covers base-sortals (subkinds, roles, and phases), Section 3.3.3 covers non-sortals (categories, roleMixins, phaseMixins, and mixins), and finally, Section 3.3.4 covers moments (relators, qualities, and modes).

3.3.1 Substantial types

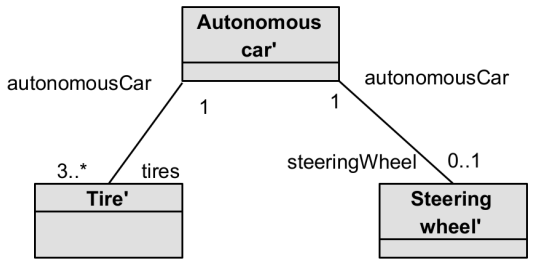
Two substantial types are covered by our transformation, namely kinds and collectives. For kinds, we also cover the scenario in which associations to other substantial types are defined.

Kinds

Kinds represent the essential things modelled within a domain [14]. Based on the UFO taxonomy, kinds are the identity providers of object types. If kinds are composed of components, they can also be called functional complexes. OntoUML does not enforce the use of specific relation stereotypes for functional complexes. We assume this is either done through a plain relation (i.e., no stereotype), or a relation marked with *componentOf* (available in the OntoUML VP plugin). This is visualised in Figure 3.2a, where an autonomous car has a *componentOf* relation to *Tire* and a plain relation to *Steering wheel*.



(A) Source model.



(B) Target model.

FIGURE 3.2: Transformation of kind as a functional complex.

In Figure 3.2b, we see the resulting implementation model. Each *kind* is transformed into a class and both relations are transformed into plain associations. We also leave out the composition marker (the black diamond), as from the perspective of the implementation, composition and aggregation relations are indistinguishable from plain associations [29][5]. Furthermore, the names of the association ends are derived from the class names (in case they are not already present in the OntoUML model). The multiplicities of the association ends stay the same.

[5]In UML, a class containing another class could mean that a class should be nested (i.e., a class declared in another class). However, we do not see this as desirable as containment in OntoUML is defined on a conceptual level, i.e., a concept is part of another concept, which would not necessarily be useful to implement as a nested class in Java.

Collectives

Collectives are atomic entities that represent a collection of individuals of the same kind. All members of a collective have the same function. For example, one can represent a band as a collective, with musicians as members. However, if one would like to represent the different roles within a band (singer, drummer, etc.) a functional complex should be used.

Another feature of collectives is that they are maximally self-contained. As an illustration, take a collective to be a group of people. In this case, the group as a whole is an entity that consists of individual people. In real life, one could say that a group of people is made up of smaller groups of people. However, this may not be said for collectives as they are non-homeorerous (relating to the concept of homeorosity). In other words, a collective is a maximally contained group that is a closed system over some social relation [18].

The *memberOf* relation stereotype indicates the members of a collective. These members are either functional complexes (i.e., kinds) or collectives. In the case of the latter, a collective is a member of another collective. An example of this would be modelling a university as a collective of faculties, where each faculty is a collective of people.

Furthermore, *memberOf* has a minimal incoming multiplicity of 2, i.e., a collective with a single member does not exist. This does not mean that this can not be represented with OntoUML (e.g., an album with a single song). For these cases where the limitations of collectives are too strict, functional complexes can be used.

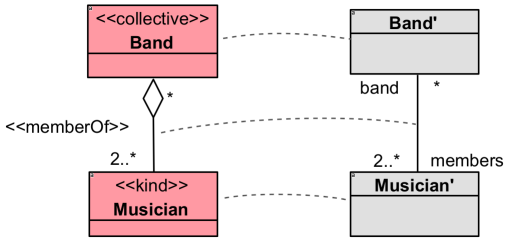


FIGURE 3.3: Transformation of Collective.

Figure 3.3 shows the proposed transformation of collectives. A Collective is transformed into a class with an association to a class representing its member type. Similar to the composition relation (visualised with the black diamond) of functional complexes, the aggregation relation (with a white diamond) is transformed into a plain association.

Furthermore, as collectives only have one type of member, the association name can simply be 'members'. Alternatively, the association name could be derived as the plural form of the member type (e.g., 'musicians' in Figure 3.3). However, that would be less trivial to derive in an automated transformation as plural terms are not always nouns with an appended 's'.

3.3.2 Base sortals

Base sortals are the types which can specialize identity providers. There are three of them, namely subkind, role, and phase. In the transformations given in this section, all base sortals specialize kinds. However, the transformations would be identical for base sortals specializing other identity providers (such as collectives, relators, or modes).

Subkinds

Subkinds are rigid sortals that specialize an identity provider. Because of their sortal nature, they can only have one identity provider as an ancestor, and therefore, they can never specialize multiple types. This means that we can take over this generalization in the implementation model without having to worry about causing multiple inheritance.

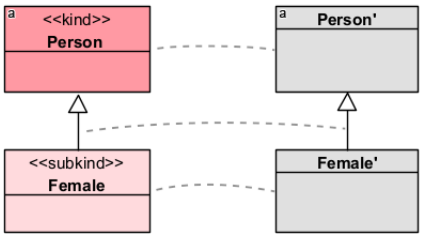


FIGURE 3.4: Transformation of Subkind.

Figure 3.4 displays the proposed transformation of subkinds. In this example, a subkind *Female* of a kind *Person* is displayed.

Roles

Roles are "relationally dependent and anti-rigid substantial universals" [16]. In OntoUML, roles are specializations of identity providers. As roles are anti-rigid, an individual of a role can cease to be in that role. This is an issue in our implementation model as in most object-oriented languages specializations are rigid [32]. Therefore, a more suitable representation is an association, which can be updated during the lifetime of an object. The chosen transformation can be seen in Figure 3.5 and is similar to the transformation defined in [32].

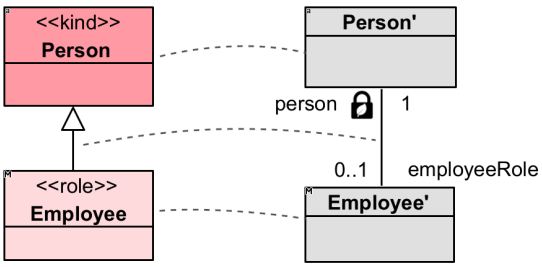


FIGURE 3.5: Transformation of Role.

The class representing a role in the resulting implementation model may change as roles are anti-rigid. However, the class representing a kind associated with a role may not change. This means that in Figure 3.5, the property *person* of *Employee* is marked as *final*.

Alternatives An alternative considered is the approach in [3] which transforms a role into a named association to a class originating from a relator (explained in more detail in Section 7.2). The benefits of this approach are that it results in fewer classes in the implementation model and it acknowledges that not the role itself, but an OntoUML *relator* type bears quality properties of a relation [16].

A caveat is that this transformation relies on a relator to be defined in the OntoUML model. One could argue that a role should always have an associated relator because roles are relationally dependent. Following this, an individual is an instance of a role only if it participates in a particular relation [16, 20, 21]. However, this is not enforced by the OntoUML VP plugin. Even more so, it is not uncommon to see roles without relators in the OntoUML model catalogue or literature (such as [20]). We assumed that it is also desirable to be able to transform these possibly less correct models.

Transforming a role into a distinct class also allows for functionality related to a role to be implemented in a dedicated location, such as a function *getAllCourses* for a role *Student*. In [3], this function would have to be implemented in the *Person* class (corresponding to the *Person* kind to which the student role belongs). This would result in the *Person* class having functions which do possibly not apply for an instance of that class. This also relates to the single-responsibility principle in object-oriented design [29], which is an argument to create two classes, one for handling all functionality relating to persons, and one for the specific functionality of students.

Furthermore, transforming a role into its own distinct class has advantages for the transformation of the roleMixin type, which is further explained in Section 3.3.3.

Phases

Phases are similar to roles in that they are anti-rigid sortals specializations of identity providers. Phases are grouped in a generalization set that is complete and disjoint. Considering the example in Figure 3.6, this means that a person is always in one state, which is either alive or dead. OntoUML itself does not include restrictions on the transitions between phases. For example, it would be allowed for a person to transition from the dead to the alive phase.

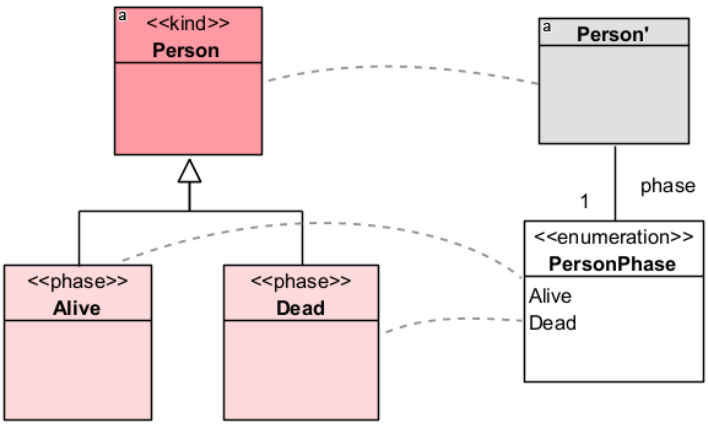


FIGURE 3.6: Transformation of a phase partition in case the phases do not have additional properties.

Two transformations for phases are given. Figure 3.6 applies to cases where phases do not have properties (termed the simple scenario) and Figure 3.7 applies when at least one of the phases has additional properties (termed the complex scenario). For the simple scenario illustrated in Figure 3.6, only the knowledge of what phase a certain kind is in seems relevant. This is simply represented by an enumeration with a multiplicity of exactly one (addressing the completeness and disjointedness). The names for the enumeration

literals are taken from the names of the phase classes.

For the complex scenario, the example in Figure 3.7 represents that a dead person has a date of death. This property can not be contained in an enumeration, so in this case, an interface is generated that is implemented by classes each representing one phase. The class representing the *Person* kind in this case then has an association with multiplicity one to the *PersonPhase* interface. Each of the classes representing a phase can then have their respective properties. In the implementation model, the relation between the class and interface is the UML *Interface realization*, which Visual Paradigm visualises as a dashed line and blue-coloured arrowhead.

The construct illustrated in Figure 3.7 resembles the state design pattern[6], which is a best practice to implement state changes. In addition, the associated *person* property of each of the phases is marked as *final* similarly to roles.

3.3.3 Non-sortals

Non-sortal types define common characteristics of the different identity providers that specialize them. Each of the four different non-sortals (categories, roleMixins, phaseMixins, and mixins) has a different set of allowed subtypes.

The non-sortal nature implies that they do not provide nor inherit a principle of identity [20]. This means that there can be no instances of non-sortal types, requiring all these to be transformed into an abstract class. Even more so, the online OntoUML documentation[7] states that all non-sortals in OntoUML are necessarily abstract. However, in the model catalogue, these are not always marked as abstract, thus we can not rely on non-sortal types being marked as abstract in OntoUML.

Another interesting feature of OntoUML is that identity providers may specialize multiple non-sortals as well as that non-sortals may extend multiple other non-sortals. This means we cannot transform a non-sortal into an abstract class and keep the generalization relations as it would result in multiple inheritance. However, Java allows the implementation of multiple interfaces (which are also non-instantiable) as well as interfaces extending multiple other interfaces. Thus, a pragmatic and Java-conformant transformation is to generate interfaces for each of the non-sortal types.

Categories

Categories are rigid non-sortals and thus provide generic characteristics for rigid types (i.e., identity providers and subkinds) [20]. The proposed transformation rule is illustrated in Figure 3.8, which shows an interface being generated for a category. The relation between *Person* and *Named Individual* is an *Interface realization*, which denotes a class implementing an interface, respectively [9].

RoleMixins

RoleMixins are a non-sortal variation of a category of which subtypes are necessarily roles [20]. The transformation rule is illustrated in Figure 3.9, in which a *Customer* roleMixin is specialised by two types of customers, following the roleMixin pattern from [19].

[6] As described by https://refactoring.guru/design-patterns/state

[7] https://ontouml.readthedocs.io/en/latest/classes/nonsortals/category/index.html

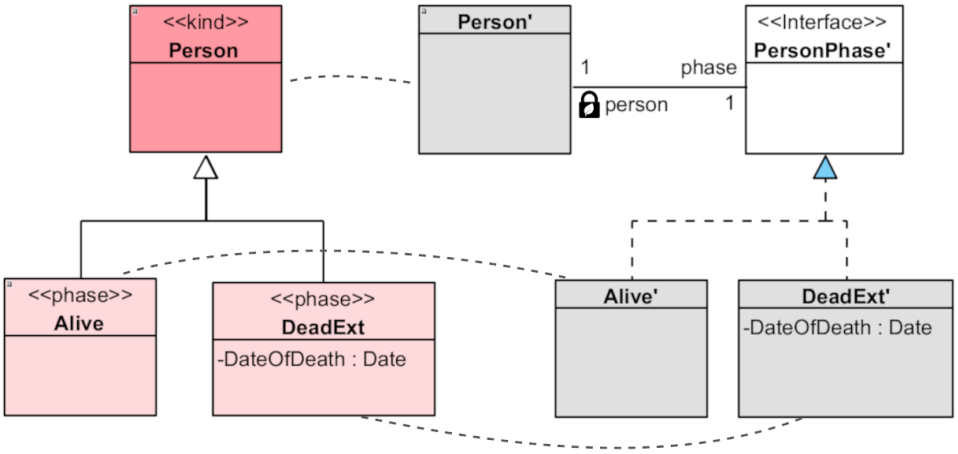


FIGURE 3.7: Transformation of phase in case one of the phases has additional properties. The relation visualised with a dashed line and blue-coloured arrowhead represents the UML *Interface Realization* relation.

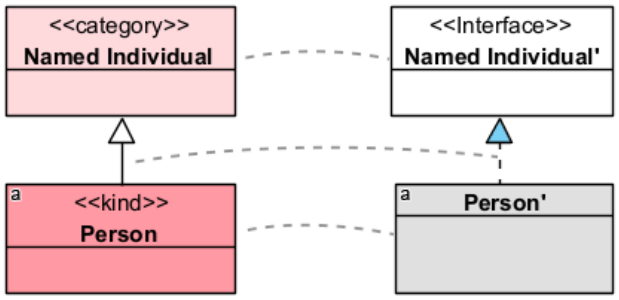


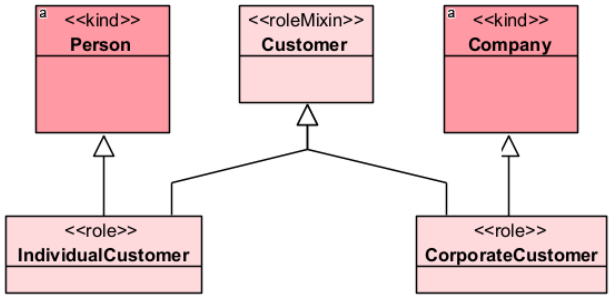
FIGURE 3.8: Transformation of Category.

Alternatives An alternative roleMixin transformation discussed in [3] was considered and is described in more detail in Section 7.2. As discussed in Section 3.3.2, the role transformation of [3] does not transform a role into its own class, therefore, there is no obvious class that represents the role in the target model of which the class representing the roleMixin can be a supertype. As a result, in [3] a roleMixin is transformed into a supertype of the classes representing the kinds of the role. For example, in Figure 3.9, *Customer* would become a supertype of both *Person* and *Company*. So all instances of *Person* and *Company* inherit features belonging to the class *Customer*, even if those instances do not instantiate *IndividualCustomer* and *CorporateCustomer*. We consider this lack of separation undesirable.

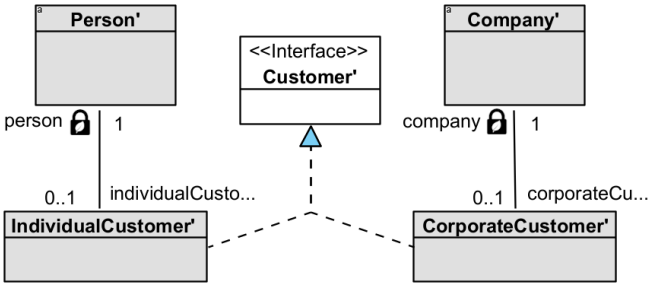
PhaseMixins

PhaseMixins provide a similar generalization for phases as roleMixins do for roles [21]. As such, the transformation rule of phaseMixins is similar to roleMixins, as illustrated in Figure 3.10.

What sets the phase transformation rule apart from the roles is that we provide two transformations for phases, the simple and complex scenario. In case a phaseMixin is present, the transformation of the complex scenario is applied. Figure 3.10a illustrates the



(A) Source model.



(B) Target model.

FIGURE 3.9: Transformation of roleMixin representing the roleMixin pattern from [19].

phase *Short-term relationship* that specializes the phaseMixin *Short-term*. This phaseMixin specialization is treated as an additional property of the *Short-term relationship* phase such that it is transformed with the complex phase transformation rule. Figure 3.10b illustrates the target implementation model including the class *Short-term relationship* that implements both the *Civil Partnership Phase* interface (corresponding to the complex phase rule in Figure 3.7) and the interface that represents the *Short-term* phaseMixin.

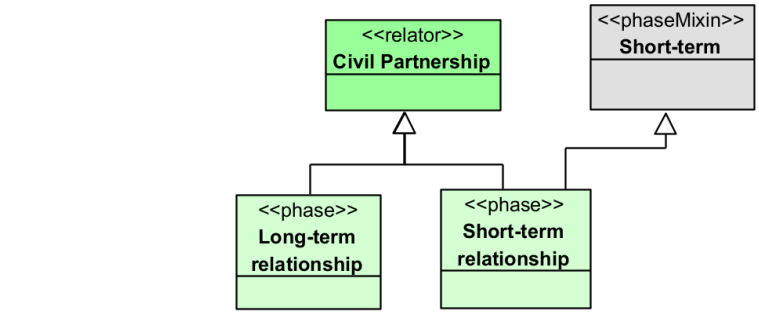
Mixins

Mixins are a special kind of non-sortal that is semi-rigid [21], so their instances can be either rigid or anti-rigid. An example of this can be seen in Figure 3.11a, in which the mixin *Performing Artist* is both a supertype of *Musician* (anti-rigid) and *Band* (rigid). In this case, a category representing *Performing Artist* would not suffice. Figure 3.11 illustrates the transformation rule which again is similar to roleMixins and phaseMixins.

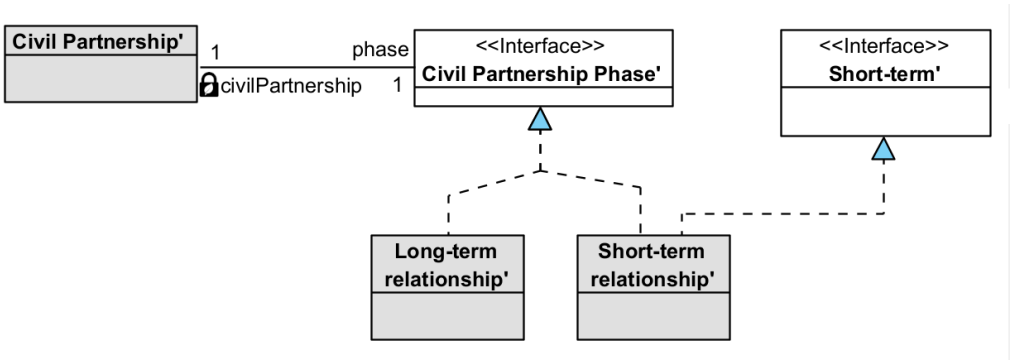
Specializations of non-sortals

Non-sortal types can be specialized by both other non-sortals or sortal types. In cases where a non-sortal specializes another non-sortal, a normal generalization is generated. In cases where a sortal specializes a non-sortal, the specialization is transformed into an interface realization. Figure 3.12 displays three examples taken from [21].

The first example illustrates a mixin specializing a category. As both types are non-sortal, the interfaces representing these types in the target model are connected by a generalization. This is similar to the second example, in which a roleMixin specializes another roleMixin, and again a generalization is generated. The third example illustrates



(A) Source model.



(B) Target model.

FIGURE 3.10: Transformation of phaseMixin.

a different case in which a kind (sortal) specializes a mixin (non-sortal). The kind *House* is transformed into a class with an interface realization towards the interface *Insured Item* representing the mixin.

3.3.4 Moment types

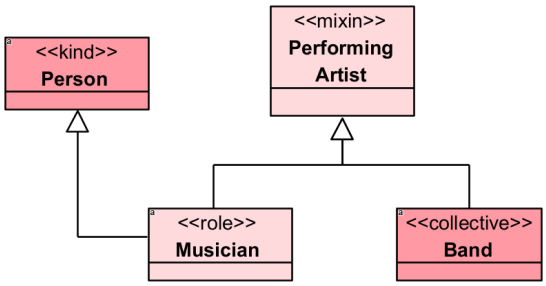
A moment, also called trope or property, is an instance that exists in another individual, which is called the bearer of that moment. As such, moments are existentially dependent on their bearer; if the bearer ceases to exist, so must the moment [16, 21, 20]. The OntoUML moment stereotypes thus represent the universals of these individual moments.

UFO distinguishes between intrinsic moments (qualities and modes) and relational moments (relators) [20].

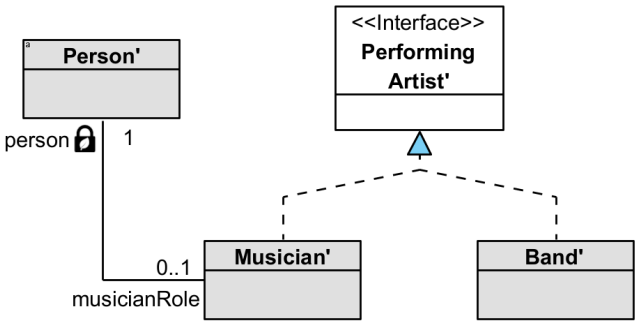
Relators

A relator bears the quality properties associated with roles [16]. They are connected through mediation relations to other classes and can be derivations of material relations. Because of their nature of relating entities, they should 'mediate' to at least two classes.

The transformation rule of the relator type is illustrated in Figure 3.13, which displays the typical relator pattern [19], where both roles are transformed as discussed in Section 3.3.2. As for all identity providers, a relator yields a distinct class. Next to this, the mediation relations are transformed into plain associations with the same cardinality. The names of the role classes are used as the association end names, respectively. Furthermore, as a relator is existentially dependent on its members [16], the associated roles of the class representing the relator may not change and are thus marked *final*.



(A) Source model.



(B) Target model.

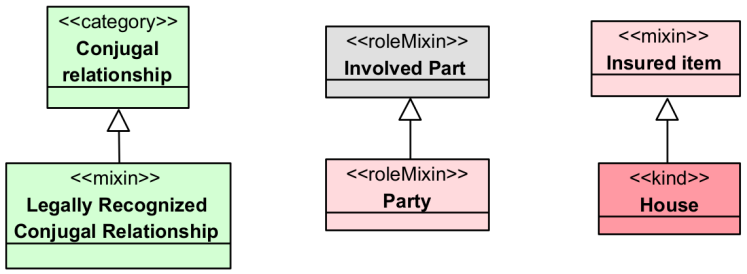
FIGURE 3.11: Transformation of mixin.

However, we decided to not represent the material relation in the target model, since this would be a duplicate access method for the opposing role class. For example, in Figure 3.13, the material relation *works for* allows the *Employee* to access its *Employer*. However, in Figure 3.13b, the *Employee* can access the *Employer* by first accessing *Employment*. When desired, a function *works for* that retrieves an employer through the employment class could be added after the transformation. However, we avoided capturing this in the implementation model.

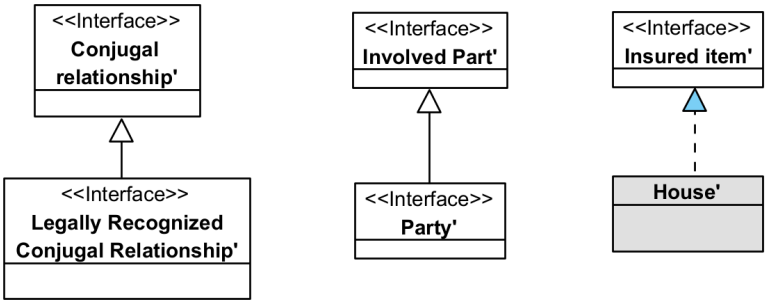
Alternatives A relator is associated with the material relation by a derivation. This construct where the relator bears the qualities of the material relation resembles a UML association class, so one could consider transforming a relator into an association class, while ignoring the mediation relations. At first glance, this might seem like a suitable solution as the material relation with its specific name is preserved and the qualities of this association are captured in the association class. However, there is no single correct way to implement an association class in Java. Rather, the final result might be something that exactly matches Figure 3.13b. Therefore, instead of moving this design decision to the implementation model to Java code generation, we addressed this in the transformation rule.

Furthermore, two other possible uses of the relator class hinder the usage of an association class. Firstly, the presence of a derivation relation associated with a material relation is not mandatory. In that case, one would be forced to use the solution illustrated in Figure 3.13b. Secondly, relators may mediate more than two classes, so an association class for a n-ary association would be required, which adds unnecessary complexity.

The transformation rule illustrated in Figure 3.13 provides a solution which seems to hold for all these possible usages of the relator type.



(A) Source model.



(B) Target model.

FIGURE 3.12: Transformation of several non-sortals specialized by other types.

Quality

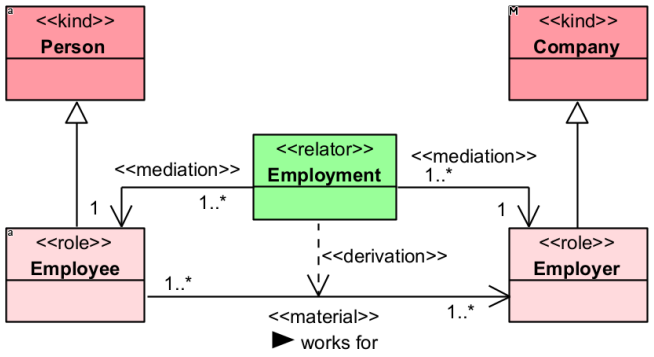
A quality is an identity provider that describes a certain property or characteristic of another type. It is extensionally dependent on its bearer, the *characterization* association stereotype indicates the relation between a quality and its bearer. This is illustrated in Figure 3.14, which visualizes the example of a flower changing colour that is depicted in [20] along with its proposed transformation.

So far, the quality type is minimally described in the UFO literature. A definition that can be found is "[qualities] are individual moments that can be mapped to some quality space, e.g., an apple's colour which may change from green to red while maintaining its identity" [21]. As such, in our transformation rule, a quality yields a class that is attached to the class representing the bearer with an association that is marked *final* in both directions.

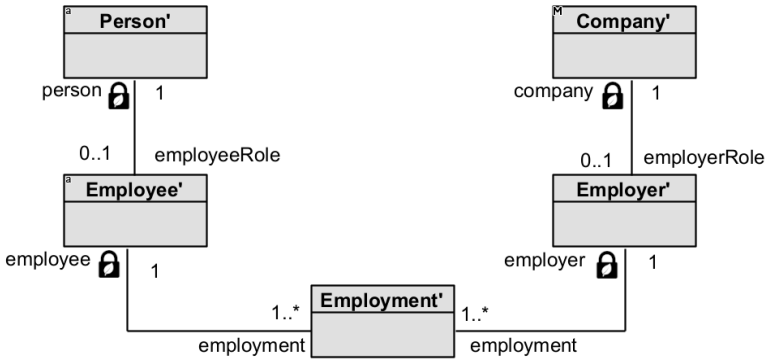
A quality is a moment that has a structured value, this value can be expressed in different units. For instance, the quality *height* of a person can be measured in meters or inches. There is no consistent way to represent this in OntoUML. The OntoUML online specification[8] refers to a special *structuration* relation stereotype that should be used for this. However, this association stereotype is not described in the literature nor available in the OntoUML VP plugin.

The characterization relation in OntoUML always points from the moment to the bearer (e.g., from *Flower colour* to *Flower* in Figure 3.14a). Furthermore, it is also uni-directional, meaning that when directly translated into code, only the moment should be able to retrieve an instance of the bearer, and not vice versa. One could argue that it is more interesting to retrieve the quality from the perspective of the bearer, e.g., we are interested in the colour of a specific flower and not per se interested in which flower belongs to a specific colour. In the transformation rule illustrated in Figure 3.14, this is mitigated by bi-directional

[8] https://ontouml.readthedocs.io/en/latest/classes/aspects/quality/index.html



(A) Source model, containing a relator pattern with roles.



(B) Target model.

FIGURE 3.13: Transformation of relator.

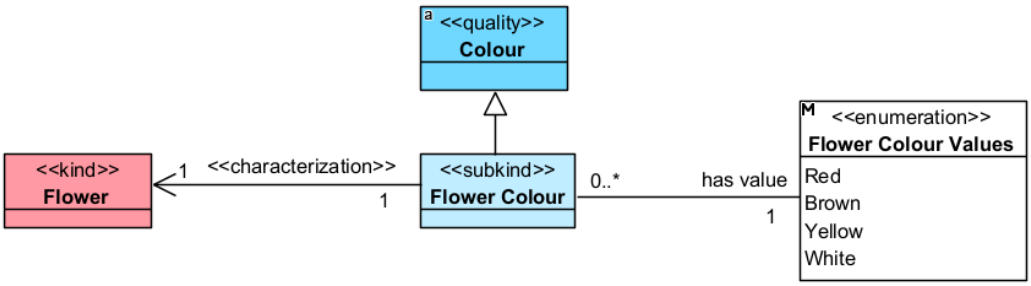
associations, i.e., the class representing the moment can access the class representing the bearer and vice versa.

Alternatives In the transformation rule illustrated in Figure 3.14, one could alternatively define an attribute in the *Flower* class that refers to the *Flower Colour* enumeration. However, an explicit choice was made to use a separate quality class assuming that the concept represented by the quality is of importance in its domain and therefore likely to require special functions to be implemented that warrant a separate class.

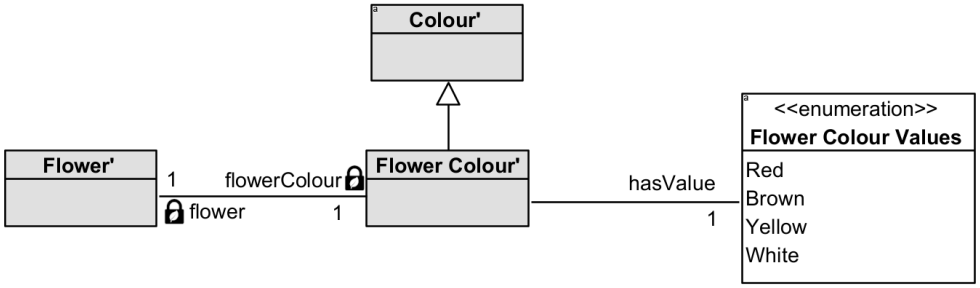
Modes

Similar to qualities, modes are not comprehensively explained in the UFO literature. An example of a mode is given in [21], which describes a person's capacity to speak a certain language. This illustrates the existential dependence of the mode on its bearer, as well as the lack of a structured value; a person either has or does not have the capacity to, for example, speak Dutch.

In Figure 3.15a, we model a sick person who has a certain disease as a mode. The characterization relation represents that the associated bearer of the moment (*Sick Person* in this case), must bear an instance of the moment. This corresponds to the lower bound



(A) Source model. OntoUML fragment taken from [20]



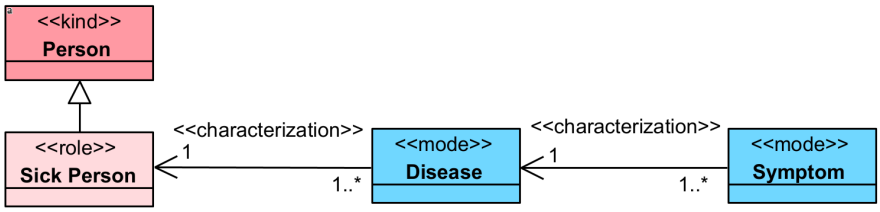
(B) Target model.

FIGURE 3.14: Transformation of a quality type that represents a flower colour.

of the cardinality at the end of the mode being one. In this example, if a certain sick person has no disease, they cease to be the *Sick Person* role.

The transformation rule of a mode, illustrated in Figure 3.15, is similar to the one of qualities in that a separate class is generated and the characterization relation is transformed into a plain association. What differs from the transformation rules of qualities is the association ends that are marked as *final*. Qualities and modes are both intrinsic properties existentially dependent on their bearer. So it is not much of a stretch to derive that the association end connected to the bearer should be marked *final*.

The other association end is more arguable. In accordance with the online OntoUML documentation and the sample ontologies provided by [20], modes can have a multiplicity of many, which seems to imply that the collection of associated modes can change. For example, for Figure 3.15a, a person can at any point in time get a new illness, or at any point in time heal of a disease by means of a disease mode that ceases to exist. For our transformation, we assume that the latter is the case, thus the class representing a mode associated with an identity provider is not marked *final*.



(A) Source model.

(B) Target model.

FIGURE 3.15: Transformation of modes.

Chapter 4

Integration of OntoUML in EMF

The first step in our transformation chain is to parse an OntoUML JSON file into a model that EMF understands. This step of the transformation chain is highlighted in Figure 4.1. This involves two artefacts: an Ecore metamodel describing OntoUML and a method to transform an OntoUML JSON file into an instance of that Ecore metamodel.

Figure 4.3 describes the OntoUML Ecore metamodel we developed. Section 4.2 describes the deviations of our Ecore metamodel compared to other existing OntoUML metamodels. Finally, Section 4.3 describes the method used to create an instance of this Ecore metamodel from an OntoUML JSON file.

4.1 OntoUML Ecore metamodel

We based the design of our OntoUML Ecore metamodel on the JSON file provided by the VP plugin through a process of metamodel discovery. Based on the models in the OntoUML model catalogue, the VP plugin is currently the leading tool for creating OntoUML models. By aligning our metamodel with the VP plugin output, any model created with the VP plugin can be used in our transformation.

The JSON provided by the VP plugin contains both model elements (relating to the abstract syntax of OntoUML) and diagram elements. The model elements describe what classes and other model elements are present in the OntoUML model as well as their properties (such as OntoUML stereotypes and attributes). The diagram elements describe how these model elements are visualised in VP, such as their shape, location, and size. For our transformation, we are not interested in the visual representation of an OntoUML model. Therefore, our Ecore metamodel only describes model elements (i.e., corresponding to the abstract syntax).

Our OntoUML metamodel is displayed in Figure 4.3. In the metamodel, every element is an instance of *OntoumlElement*, which gives each element a unique id, a name, and possibly a description. The top element in an OntoUML model is a *Project*, which contains all elements of a model. Besides *Project*, all other metamodel elements are an instance of *ModelElement*. Each *ModelElement* is contained in exactly one package. A *Package* is also a *ModelElement*, meaning that packages can be contained in other packages. Each OntoUML model has one super *Package* that is contained directly in the *Project*.

We describe the main elements in the Ecore metamodel with the use of a small example model, which is illustrated in Figure 4.2. This model is annotated with references to the metamodel elements.

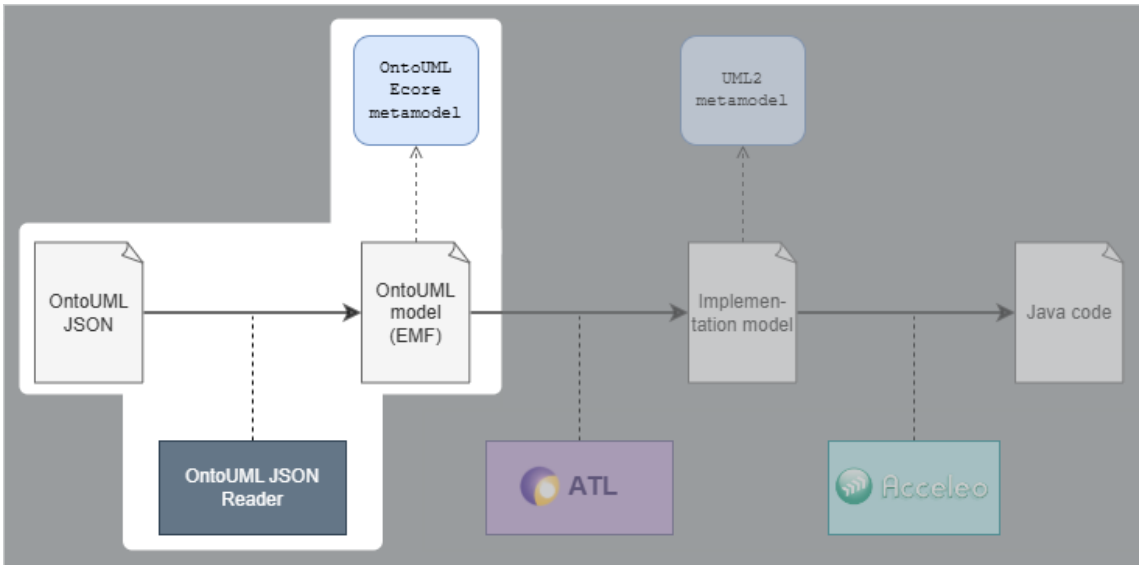


FIGURE 4.1: Transformation chain that highlights the step of parsing an OntoUML JSON file to an Ecore model.

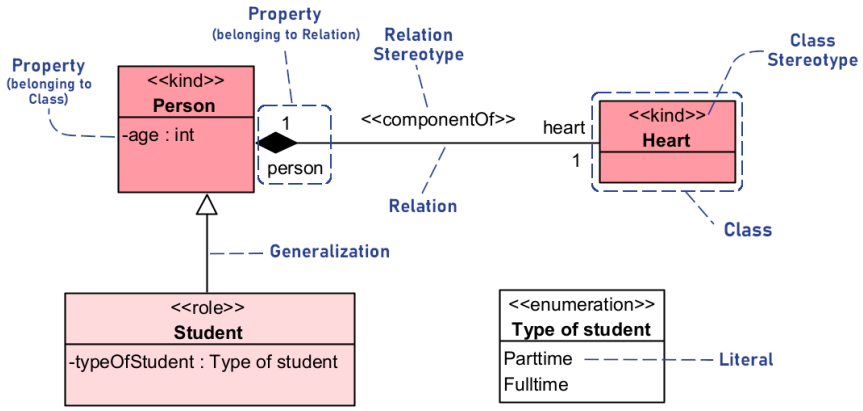


FIGURE 4.2: OntoUML diagram annotated with references to the OntoUML metamodel.

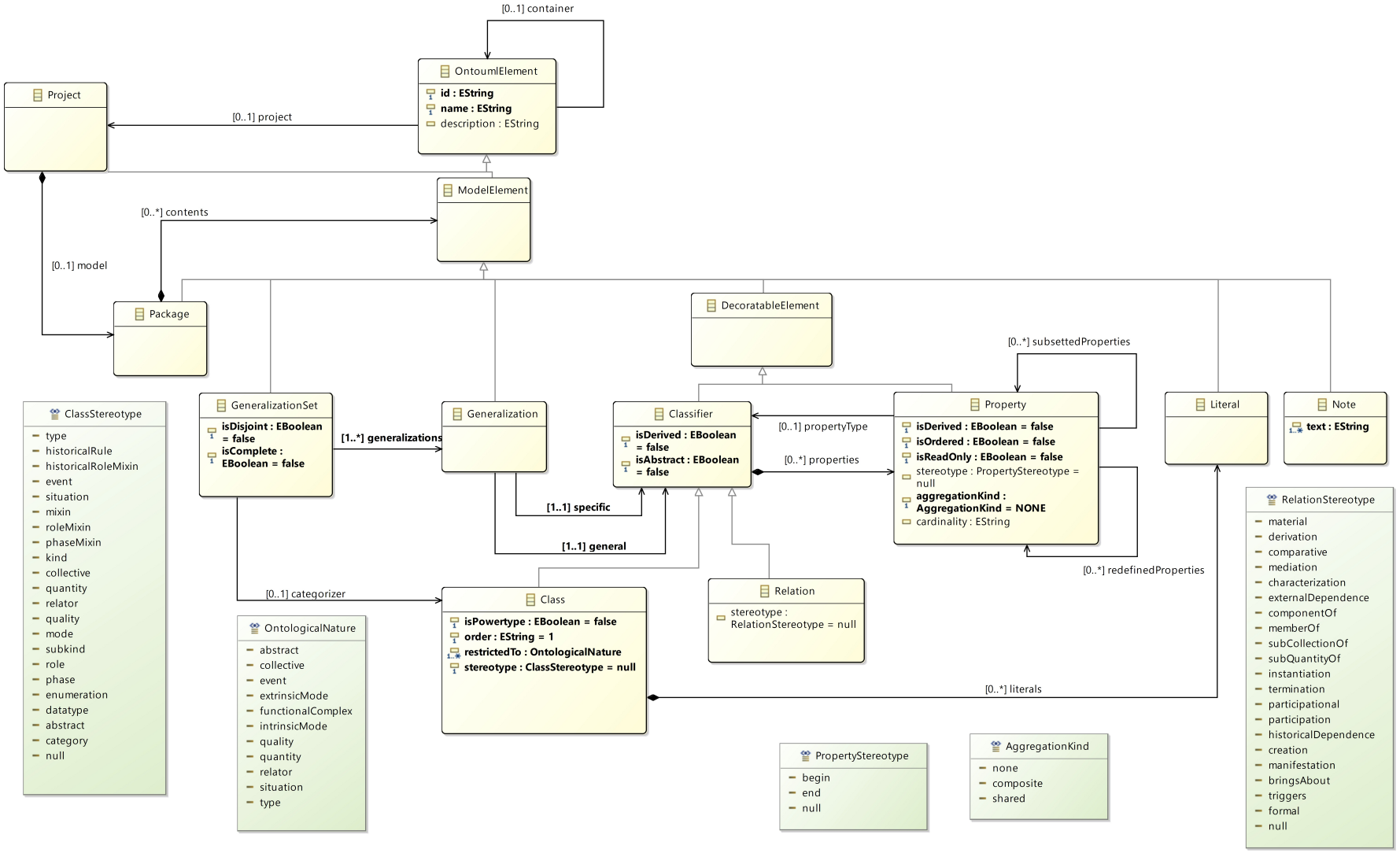


FIGURE 4.3: OntoUML Ecore metamodel

4.1.1 Classes

Figure 4.2 shows an OntoUML model represented as a UML class diagram with OntoUML types added. Each class has a *ClassStereotype*, which in this case are three UFO concepts (*kind* and *role*) and one generic UML stereotype (*enumeration*).

An enumeration is a special kind of class. As opposed to other classes, an enumeration may have literal values (i.e., the *Literal* model element). The *Type of student* enumeration of Figure 4.2 has two literal values, namely *Parttime* and *Fulltime*[1].

4.1.2 Relations

An OntoUML relation connects different classes. In the metamodel, the classes connected by a relation are included in the reference called *properties*. Similar to classes, they may have a stereotype, which is one of the values from *RelationStereotype*. In Figure 4.2, a relation with the *componentOf* stereotype can be seen that connects the *Person* to the *Heart* class.

4.1.3 Properties

Properties are references to other classifiers in an OntoUML model. The classifier to which they refer is included in *propertyType*. Properties are contained in classifiers, which means that both *Class* and *Relation* may have properties. In Figure 4.2, we see two examples of class properties, both displayed inside classes with a prepended dash. The property type is displayed after a colon, such as the type *int* for the property *age* of *Person*. In UML (and Visual Paradigm), *int* refers to a primitive datatype which is not visually displayed in the model. Property *typeOfStudent* has as type *Type of student*, which is a classifier in the same model.

In relations, properties refer to the classes that are connected by the relation. The *componentOf* relation displayed in Figure 4.2 has two properties. The highlighted one is named *person* and has the *Person* class as type. This property also has the *aggregationKind* *composite*, represented with a black diamond.

All properties have a cardinality attribute. The relation properties *person* and *heart* displayed in Figure 4.2 both have a cardinality *1*. Class properties also have a cardinality, although these are not visualised in Visual Paradigm. The default cardinality of these class properties is *1*.

4.1.4 Generalizations

A generalization in the OntoUML metamodel has two attributes, namely *specific* and *general*. These attributes indicate that the class referred to by *specific* specializes the superclass, referred to by *general*. As such, generalizations are also called specializations.

In Figure 4.2, a generalization connects *Student* (the *specific*) with *Person* (the *general*), while the latter is indicated with the arrowhead.

[1]The metamodel does not restrict other classes to have enumeration literals. However, Visual Paradigm only allows literals to be defined in enumerations, which should also be enforced by other OntoUML model editor tools.

4.2 Differences with existing OntoUML metamodels

In the OntoUML tool suite, two projects exist that also describe the OntoUML JSON structure. The first is the OntoUML JSON Schema[2], which should describe the structure to which the OntoUML JSON should adhere. However, this schema deviates from the JSON files the OntoUML VP plugin generated and therefore was of no use to us when defining the Ecore metamodel. The second is the platform-independent metamodel[3], which serves as a general reference of the OntoUML structure independent of the technology used. A summary of the abstract syntax of this model is displayed in Figure 4.4. Although this metamodel is better aligned with the JSON output of the VP plugin, it still differs in some aspects. We used this platform-independent metamodel as inspiration for the creation of the Ecore metamodel. In this section, we describe the deviations between our Ecore metamodel (which is matched to the VP plugin) and the platform-independent metamodel.

These differences can be divided into two categories:

1. Differences due to the metamodels being defined in different modelling languages, since the platform-independent metamodel is defined in UML whereas our Ecore metamodel is defined in Ecore.

2. Differences required to align our OntoUML metamodel with the output of the OntoUML VP plugin.

The second category of differences exposes where the platform-independent metamodel deviates from the VP plugin and thus may aid in future work aligning different OntoUML tools.

4.2.1 Differences due to different metametamodels

As stated, the platform-independent metamodel and our Ecore metamodel are defined in different syntaxes. This is illustrated in Figure 4.5, where the platform-independent metamodel sits on the M1 level on the left-hand side, and our OntoUML metamodel sits on the M2 level on the right-hand side.

UML and Ecore are similar in that they both consist of classes with attributes and associations. However, there are some constructs in UML used by the platform-independent metamodel that are not directly available in Ecore.

No non-resolved datatypes

Two datatypes are defined in the platform-independent metamodel, namely *LanguageString* and *Cardinality*. Although datatypes are available in Ecore metamodels, they refer to pre-existent/implemented Java classes. This could be useful if, for example, one would like to include a date attribute. In that case, one could use a datatype that points to the Date class provided by the standard available Java util library[4].

Our Ecore metamodel does not include either *LanguageString* nor *Cardinality*, because they are not used as such by the OntoUML VP-plugin, as described in Section 4.2.2. However, in case the OntoUML VP plugin is updated to mitigate this, these datatypes can be included as normal classes.

[2]See https://github.com/OntoUML/ontouml-schema/blob/master/src/ontouml-schema.yaml#L8.
[3]See https://github.com/OntoUML/ontouml-metamodel.
[4]See https://docs.oracle.com/javase/8/docs/api/java/util/Date.html.

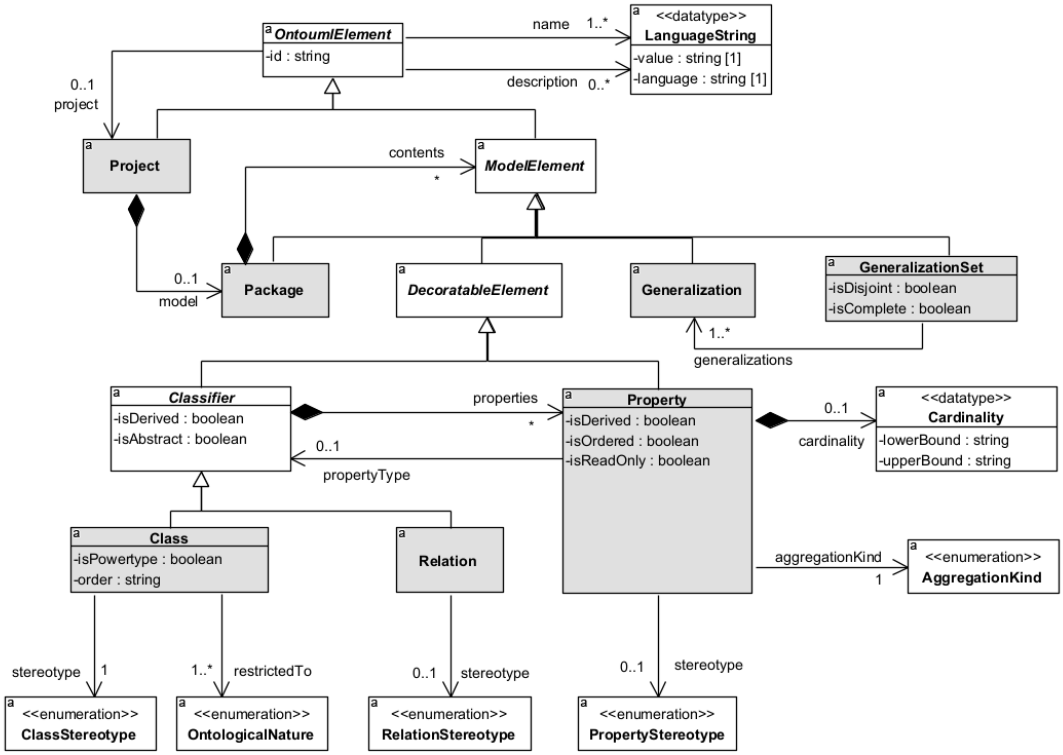


FIGURE 4.4: Summary of the platform-independent OntoUML metamodel.

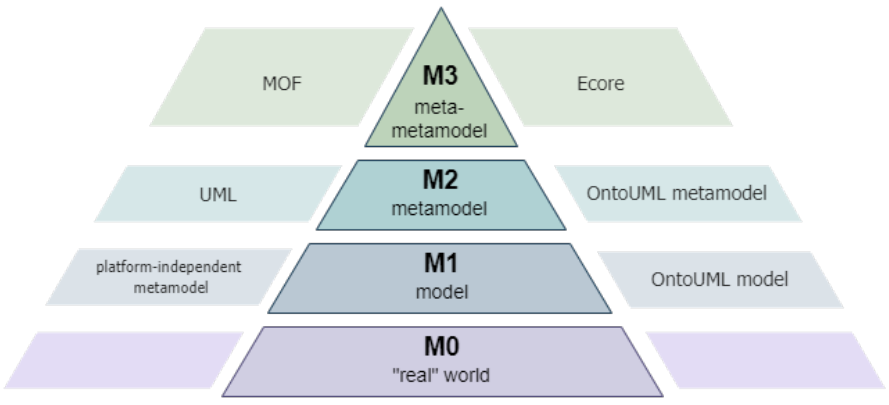


FIGURE 4.5: Metamodel hierarchies of the OntoUML platform-independent metamodel (left-hand side of the pyramid) and the EMF OntoUML metamodel (right-hand side).

Enumeration references

While the enumerations in the OntoUML metamodel are included by references, ECore only allows enumerations to be class attributes. So instead of *Class* having an association to the enumeration *ClassStereotype* with the name *stereotype*, this stereotype is included as an attribute with the corresponding type.

Inclusion of extra enumeration literal 'null'

Attributes in Ecore have a mandatory default value. For attributes that have an enumeration as type, this default value cannot be *null*. This is even true for attributes with a lower bound of 0, which theoretically should be allowed to have no value.

When parsing a JSON file into an instance of an Ecore metamodel, attributes of an enumeration type with the JSON value *null* would result in the default value being used (which is the first enumeration literal if not explicitly set). Therefore, if an OntoUML model contains the *null* value for an enumeration attribute, it can not be derived if a stereotype was originally *null* or whether it was explicitly set to the default value. To mitigate this, an extra *'null'* literal was added to each of the enumerations, which is also explicitly set as the default value.

The only exception to this is the enumeration *OntologicalNature*, which is included in *Class* with a multiplicity of 1-to-many. Hence, the *null* value in a JSON file is mapped to an empty list.

Enumeration literal names

The platform-independent metamodel makes use of names with dashes for the OntologicalNature enum. Enumeration literals names in Ecore are not allowed to have dashes. Therefore, these names are transformed into camelcase. The literal value of the enum values is set to the dash-variant (as displayed in Figure 4.6) to stay compatible with the JSON file. The values of the other enums (i.e., *ClassStereotype* and *RelationStereotype*) are in camelcase in the platform-independent metamodel. Therefore these do not have to be altered.

4.2.2 Differences in metamodel due to output of OntoUML VP-plugin

At the moment of writing, the JSON output of the OntoUML VP-plugin is inconsistent with the platform-independent metamodel. To facilitate the parsing of OntoUML JSON files to EMF, all the Ecore types, names, and cardinalities should match that of the JSON files.

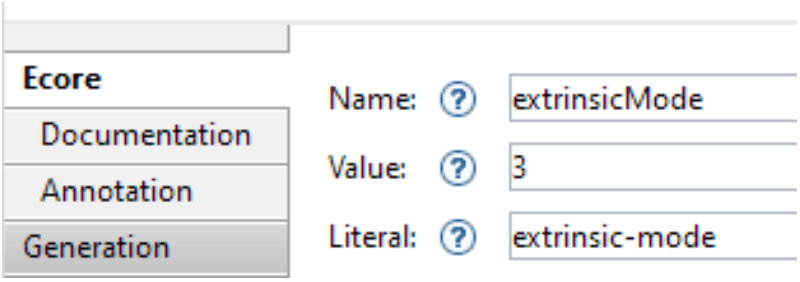


FIGURE 4.6: OntologicalNature enum value extrinsic mode with camelcase name and dashed-named literal.

The following deviations from the platform-independent metamodel were necessary to align the Ecore metamodel with the VP-plugin JSON file.

Missing values for LanguageString and Cardinality

The platform-independent metamodel contains a datatype for *LanguageString*, which has two string attributes for a value and language. Supposedly, this is to support model elements with text values in multiple languages. For example, a class could have as name the *LanguageString {value: "Car", language: "en"}* as well as *{value: "Auto", language: "nl"}*. However, the OntoUML VP-plugin disregards this info and simply outputs the language string of the preferred language[5]. Therefore, in the Ecore metamodel, *LanguageString* is disregarded and in these cases it is replaced by a string attribute with multiplicity *1*.

There is a similar discrepancy for the datatype *Cardinality*. Whereas the datatype has two attributes *lowerBound* and *upperBound*, the VP-plugin stores the cardinality in the form which is displayed in a UML diagram, such as *"2..*"*, *"1"*, or *"*"*. Therefore, in our Ecore metamodel, the cardinality of a property is represented as a string.

Name of 'literal' relation for Class

In the platform-independent metamodel, the Class type has a relation 'literal' with cardinality *0..** to the *Literal* class. However, the VP-plugin uses the plural form 'literals'.

Capitalised enum values for AggregationKind

The AggregationKind enum in the platform-independent model has the values *none*, *composite*, and *shared*. However, in the VP-plugin, the values are capitalised. This is accommodated by capitalising the literal value of these enum values, as is shown in Figure 4.7.

Project as an OntoUmlElement

In the VP-plugin, a project extends the OntoUmlElement. As such, a project also has a name and description in the JSON. To include these values, the *Project* class extends the *OntoUmlElement* in our Ecore metamodel.

Property assignments of ModelElement

The VP-plugin has an attribute *propertyAssignments* for the *ModelElement* class, which contains a Java Map with Strings as keys and Objects as values. Since there is no documentation on the use of this attribute, we decided to ignore this attribute in the Ecore metamodel.

4.3 OntoUML JSON Reader

Under the hood, EMF is a Java library that contains a model structure for EMF objects, or EObjects as they are called. An Ecore metamodel is transformed into Java code by

[5] In the implementation of the VP plugin, a class LanguageString class is used. However, if only one language string is present for a model element (which is the case for all observed models), the exported JSON only includes a single plain string. Unfortunately, there is no easy way to work around this issue of an attribute with the same name that has different forms in Ecore. It is worth noting that this behaviour of the VP-plugin both contradicts the JSON schema as well as the metamodel.

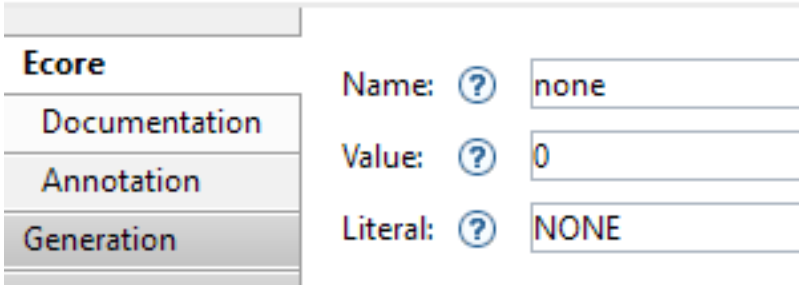


FIGURE 4.7: Value of the AggregationKind none.

extending and implementing these EObjects, so that an instance of an Ecore metamodel is a collection of Java objects instantiating the EObjects of the corresponding metamodel. In EMF, such a collection of EObjects is called a resource.

The default method of persisting resources in EMF is through the XMI format. However, as OntoUML uses JSON files for interchanging ontologies, we should be able to generate an Ecore model from these OntoUML JSON files. In this way, an OntoUML model created with any of the OntoUML tools (such as the VP-plugin) can be used within the EMF tool suite.

The EMFJSON-Jackson library[6] can be used to persist Ecore metamodels in the JSON format. This library is an adapter for the Java Jackson library, which is the de facto standard in Java for JSON (de)serializing. In Jackson, a custom object mapper can be used that maps Java objects and attributes to JSON nodes. Alternatively, Jackson provides Java annotations that can be attached to Java classes to make this process less cumbersome.

When using Jackson for EMF, it would be necessary to define a custom object mapping for the Ecore metamodel,either through an Object Mapper or Jackson annotations, with the consequence that this object mapping has to be updated each time the Ecore metamodel changes. EMFJSON provides a generic solution to this problem by using the generated model factories to instantiate an instance of an Ecore metamodel.

4.3.1 Differences between EMFJSON and OntoUML

Although EMFJSON provides a generic solution, there are differences in how EMFJSON and OntoUML store type and reference information in JSON files. A small sample model will be used to illustrate the differences between OntoUML JSON and the default EMFJSON representation. This model, displayed in Figure 4.8, contains a containment reference (from the computer to its components), a reference relation (from a CPU to a motherboard, to which it is connected), and an identity provider for components (the *id* field; which is marked as an identifier).

The EMFJSON library was used to store an instance of this model in JSON format, listed in Listing 4.1. In this JSON snippet, we see the field *EClass* containing the URI of the model used (in this case an example.org URL) followed by the Ecore type of the object. The containment reference is represented by the respective objects being child nodes. For the reference relation of *CPU*, we see that the EClass of the relation is included (i.e., *Motherboard*) along with the respective id in the *ref* field.

In Listing 4.2, we see a fragment of OntoUML JSON that contains a generalization (as also present in the metamodel visualised in Figure 4.3). From this, some differences with EMFJSON representation become apparent. First of all, the type of the object is

[6]See https://github.com/eclipse-emfcloud/emfjson-jackson.

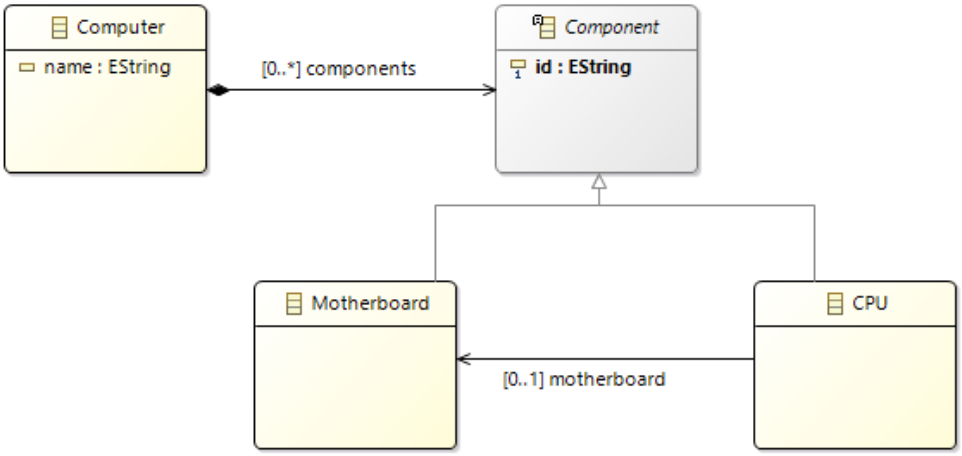


FIGURE 4.8: Sample metamodel in Ecore representing a computer with components.

```
{
  "EClass" : "http://www.example.org/referenceSample#//Computer",
  "components" : [ {
    "EClass" :
    ↪ "http://www.example.org/referenceSample#//Motherboard",
    "id" : "A"
  }, {
    "EClass" : "http://www.example.org/referenceSample#//CPU",
    "id" : "B",
    "motherboard" : {
      "EClass" :
    ↪ "http://www.example.org/referenceSample#//Motherboard",
      "$ref" : "A"
    }
  } ]
}
```

LISTING 4.1: Default JSON representation of the EMFJSON-Jackson serializer.

contained in the *type* field (as opposed to *EClass*). Furthermore, the type solely consists of the class name and so does not include the Ecore metamodel URI, which is not surprising as the OntoUML model originally was not an EMF model. Lastly, the references consist of the field *id* and *type* instead of *ref* and *EClass*.

Luckily, EMFJSON allows for the customization of these properties. We implemented a custom EMFJSON Module so that the EMFJSON library can be used to deserialize OntoUML JSON into the EMF representation[7].

[7]The code of this custom mapper can be found at https://github.com/GuusVink/ontouml-java-generation.

```
...
{
  "id" : "X2BpewmGAqAEFQ_g",
  "name" : null,
  "description" : null,
  "type" : "Generalization",
  "propertyAssignments" : null,
  "general" : {
    "id" : "tpWpewmGAqAEFQ_H",
    "type" : "Class"
  },
  "specific" : {
    "id" : "faupewmGAqAEFQ_U",
    "type" : "Class"
  }
  ...
```

Listing 4.2: Fragment of OntoUML JSON representing a generalization which contains two references.

Chapter 5

Transformation implementation

This chapter describes the transformation from an OntoUML model in EMF to Java code. Section 5.1 describes the first step that transforms an EMF OntoUML model into an implementation model using ATL. Section 5.4 describes how the implementation model is transformed into Java code using Acceleo. Section 5.5 describes how the transformation chain as a whole can be executed outside the Eclipse environment.

5.1 Transformation implementation in ATL

We have implemented our proposed transformation of Section 3.3 in ATL. Besides the OntoUML type transformations discussed in Section 3.3, our transformation also covers class properties, enumerations and datatypes. Figure 5.1 illustrates the transformation chain of the entire transformation and highlights the step of the ATL transformation.

Figure 5.2 illustrates the transformation signature of the ATL transformation: the transformation takes a model adhering to our OntoUML metamodel and generates an implementation model defined by UML. The UML2 metamodel refers to the EMF UML2 implementation[1].

Our transformation consists of a main module, containing the main transformation logic, and two utility libraries that contain helper functions to facilitate the main transformation. We first discuss the two utility libraries *MyStrings* and *OntoUmlUtilities*, then we provide a high-level description of our main transformation along with relevant design decisions made. Our transformation assumes that the provided OntoUML model is 'correct'. Besides complying with our OntoUML Ecore metamodel, we provide some additional constraints relating to the OntoUML semantics that we expect a model to adhere to, which are listed at the end of this section.

5.1.1 Utility libraries

Besides the main module, we defined two utility libraries, namely *MyStrings* and *OntoUmlUtilities*. These are both imported and used in the main ATL module. These utility libraries contain ATL helper functions that support the main transformation logic. By having these in separate files, we aim to not compromise the readability of the main transformation logic.

[1] See https://projects.eclipse.org/projects/modeling.mdt.uml2.

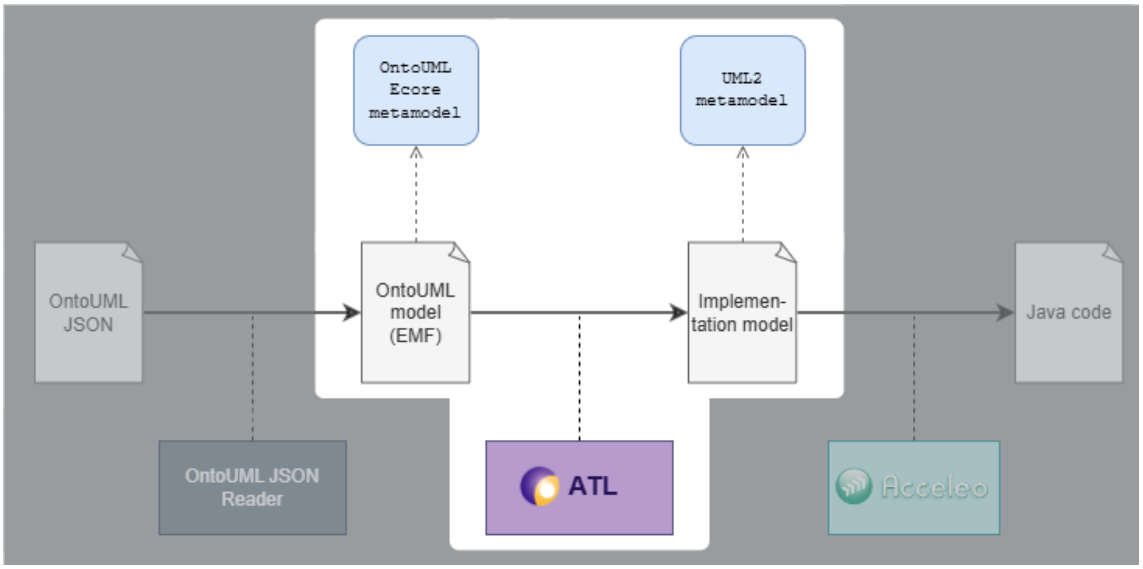


FIGURE 5.1: Transformation chain that highlights the step going from an OntoUML model to an implementation model.

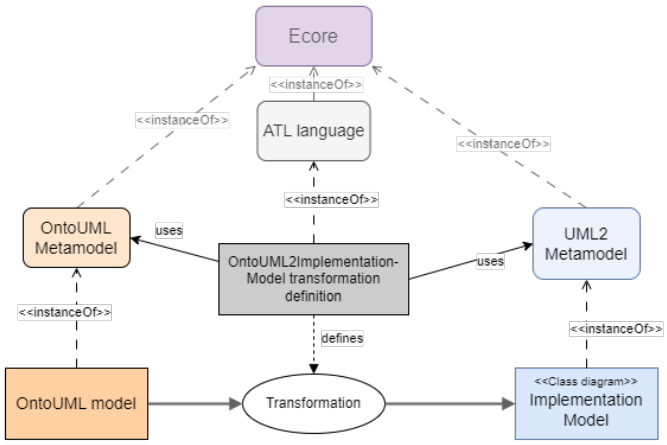


FIGURE 5.2: Transformation pattern of the OntoUML-to-implementation-model transformation.

MyStrings

The *MyStrings* library provides some of the required string operations that are not present in OCL. We chose the name *MyStrings* to indicate that it is different from the *strings* library, which was a standard library in older versions of ATL that is not available anymore.

Transforming strings into valid names Three helper definitions transform arbitrary strings into strings that are valid in the Java programming language and are used to generate names for classes, methods/attributes, and enumeration literals. In the current implementation, the Java naming conventions are used[2]. This means that class names are transformed into *PascalCase*, method and attribute names into *camelCase*, and enumeration literals into *UPPER_SNAKE_CASE*.

Furthermore, special characters that may not occur in Java identifiers are removed. Allowed characters are letters, digits, '_' and '$'[3]. To determine what characters are valid letters, the unicode character category 'L' is used[4]. This means that special letters such as 'é', 'ç', and 'ã' are also supported.

Unpacking cardinalities As discussed in Section 4.2.2, the OntoUML Ecore model stores cardinalities in a single string, such as 1..∗, ∗, or 2..4. The UML model requires properties to have a separate lower and upper bound. Two helper definitions are implemented to extract both bounds from the OntoUML cardinality.

OntoUmlUtilities

The *OntoUmlUtilities* library contains helper definitions specifically for OntoUML elements. It performs relatively more complex queries on the OntoUML model to detect larger structures. For example, it contains functions to check whether an identity provider has phases associated with it, and whether these phases contain properties or not. These utility functions are used to determine whether a phase should be transformed according to the simple or complex scenario, as described in Section 3.3.2.

Furthermore, helper functions for properties are included that determine what type of relations they belong to. This is relevant for the transformation of some OntoUML relation types such as *mediation* and *memberOf*, as the properties of these relations are treated differently from normal relation properties.

5.1.2 Main module

This section discusses the main design decisions and relevant implementation details. We do not discuss the entire transformation, which can be found at https://github.com/GuusVink/ontouml-java-generation. For illustration purposes, we include some fragments of the transformation definition.

Global structure

Our ATL transformation makes use of rule inheritance to structure the rules for different OntoUML model elements. The rule *ModelElement2PackageableElement* listed in Listing 5.1 is located at the top of this hierarchy, as summarised in Figure 5.3. This rule is

[2]See https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html.

[3]As per the Java language specification, see https://docs.oracle.com/javase/specs/jls/se21/html/jls-3.html#jls-3.8.

[4]See https://www.regular-expressions.info/unicode.html#prop.

extended by rules that generate various UML elements such as associations, classes, and packages. The rule itself is abstract, meaning that it generates no target elements itself.

```
abstract rule ModelElement2PackageableElement {
  from element : OntoUML!ModelElement
  to pe : UML!PackageableElement()
}
```

LISTING 5.1: Top rule in the rule hierarchy that defines the transformation of OntoUML ModelElements to UML PackageableElements.

The main purpose of this rule hierarchy is that it enables the use of the ATL *resolveTemp* function. The *resolveTemp* function allows to point to target model elements that have been generated from a specific source model element. A small example can be seen in the *Package2Package* rule listed in Listing 5.2. In OntoUML, a *Package* contains *ModelElements* reachable through the variable *contents*, shown in Figure 4.3. In the *Package2Package* rule, the elements from an OntoUML Package are taken, and the *resolveTemp* function is used to collect all UML elements that were generated from these OntoUML *ModelElements* through the *ModelElement2PackageableElement* rule, independent of what type of PackageableElement was generated. The *'pe'* string refers to the name of the generated element defined in the super rule *ModelElement2PackageableElement* listed in Listing 5.1.

```
rule Package2Package extends ModelElement2PackageableElement {
  from element : OntoUML!Package
  to pe : UML!Package (
    name <- thisModule.toPackageName(element.name),
    packagedElement <- element.contents->collect(e | thisModule.resolveTemp
    (e, 'pe'))
  )
}
```

LISTING 5.2: Rule for transforming OntoUML Packages into UML Packages.

An abstract subrule of *ModelElement2PackageableElement* is *Class2Classifier*, which handles the generation of UML classifiers such as classes, enumerations and interfaces. The first rule extension of this rule is the *IdentityProvider2Class* rule, which generates a UML Class for all OntoUML identity providers (i.e., whether the stereotype of that OntoUML class is present in a list of identity provider stereotypes).

Other *Class2Classifier* rules are present for datatypes and enumerations, which are discussed in more detail in Section 5.1.2, and the generation of interfaces for non-sortal OntoUML types.

Transformation of phases As discussed in Section 3.3.2, the transformation of phases is split into two cases: the simple and complex scenario. To accommodate for this, we defined two rules, one for each scenario respectively, that extend the *IdentityProvider2Class* rule, where one rule checks whether the simple scenario applies and generates UML elements according to the transformation displayed in Figure 3.6, and the other rule checks whether the complex scenario applies and generates UML elements according to Figure 3.7.

Transformation of relations The *Relation2Association* rule transforms OntoUML relations into UML associations. This rule first checks whether the stereotype of a relation is

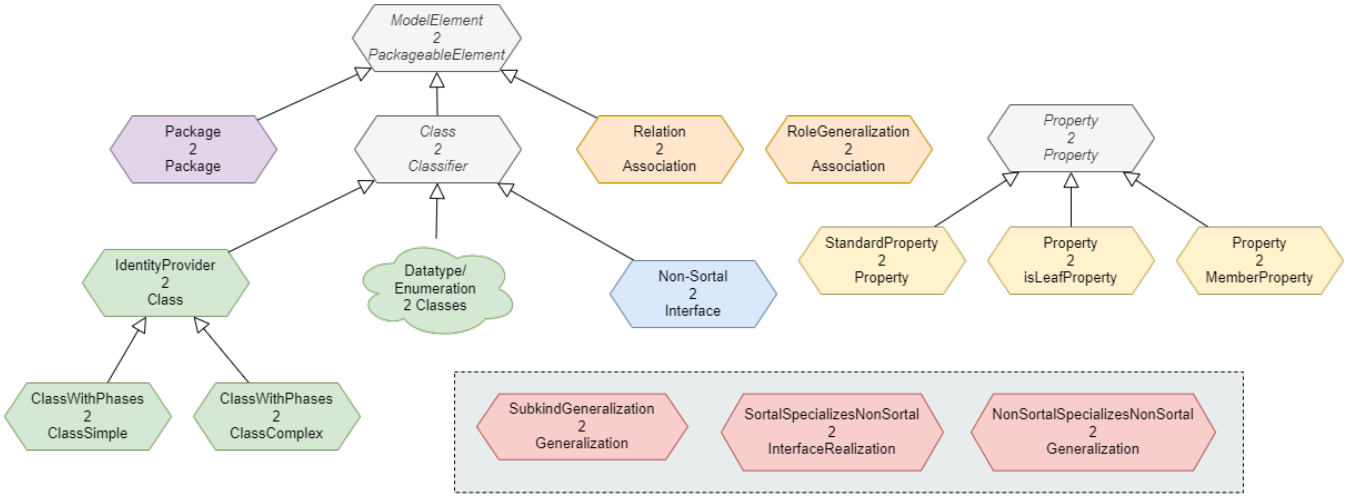


FIGURE 5.3: Summary of rules (and their inheritance) in the OntoUML-to-ImplementationModel ATL transformation.

supported by the transformation. If that is the case, the relation is treated as described in the transformation of their corresponding class stereotype, such as the transformation of *collectives* for the *memberOf* relations, or the transformation of *relators* for the *mediation* relations.

Furthermore, we defined a rule for transforming an OntoUML Generalization between a role and identity provider into an association, conforming to the transformation described in Section 3.3.2.

The transformations described in Chapter 3 treat some OntoUML relations differently from others. For example, one end of a *mediation* relation is marked as final by setting the UML *isLeaf* attribute (as depicted in Figure 3.13). The logic behind these transformation mechanisms is defined in the *Property2Property* rules. Depending on the relation type a property is part of, either a normal UML Property or a property with the attribute *isLeaf* set to true is generated. A third property rule generates a UML Property with the name '*members*' for *memberOf* relations of collectives, as discussed in Section 3.3.1.

Transformation of generalizations Besides the *RoleGeneralization2Association* rule, Figure 5.3 shows three other rules that transform OntoUML generalizations, coloured in red. One of these is the transformation of generalizations that involve subkinds, of which the code is listed in Listing 5.3.

Firstly, in the *from* section, the rule checks whether the generalization corresponds to a subkind specializing either an identity provider or another subkind. These are considered the only valid uses of subkinds.

In the *to* section of *SpecializationSubkind*, we see another usage of the *resolveTemp* function in combination with the used rule hierarchy. The *general* attribute from a UML Generalization is set to the UML Classifier that is generated from the OntoUML Class referenced in the *general* attribute of the to-be-transformed generalization. This same mechanism is used for the *specific* attribute in Listing 5.3.

```
--- Transformation of subkind specialization
rule SubkindGeneralization2Generalization {
  from g : OntoUML!Generalization (
    -- Subkind may specialize identity providers or other subkinds (
    explitly not non-sortals)
    g.specific.stereotype = #subkind
    and thisModule.identityProviders->append(#subkind)->includes(g.general.
    stereotype)
  )
  to gen : UML!Generalization (
    general <- thisModule.resolveTemp(g.general, 'pe'),
    specific <- thisModule.resolveTemp(g.specific, 'pe')
  )
}
```

LISTING 5.3: Matched rule that transforms generalizations of subkinds into generalizations in the target model.

Transformation of enumerations, datatypes, and class properties

Besides all the OntoUML types that are covered by our transformation design, our transformation also transforms enumerations, datatypes, and class properties. Figure 5.4 shows an example of an OntoUML model with these elements along with the generated UML model.

In OntoUML, both enumerations and datatypes are normal classes with a respective *enumeration* or *datatype* stereotype. Enumerations are trivially transformed into UML Enumerations, while datatypes are transformed into plain classes.

Class properties Class properties primarily consist of a name, type, and cardinality. The type may refer to either a UML Primitive Type or another class (including enumerations and datatypes). The supported primitive types in our transformation are *int*, *boolean*, *string*, *float*, *char*, and *double*.

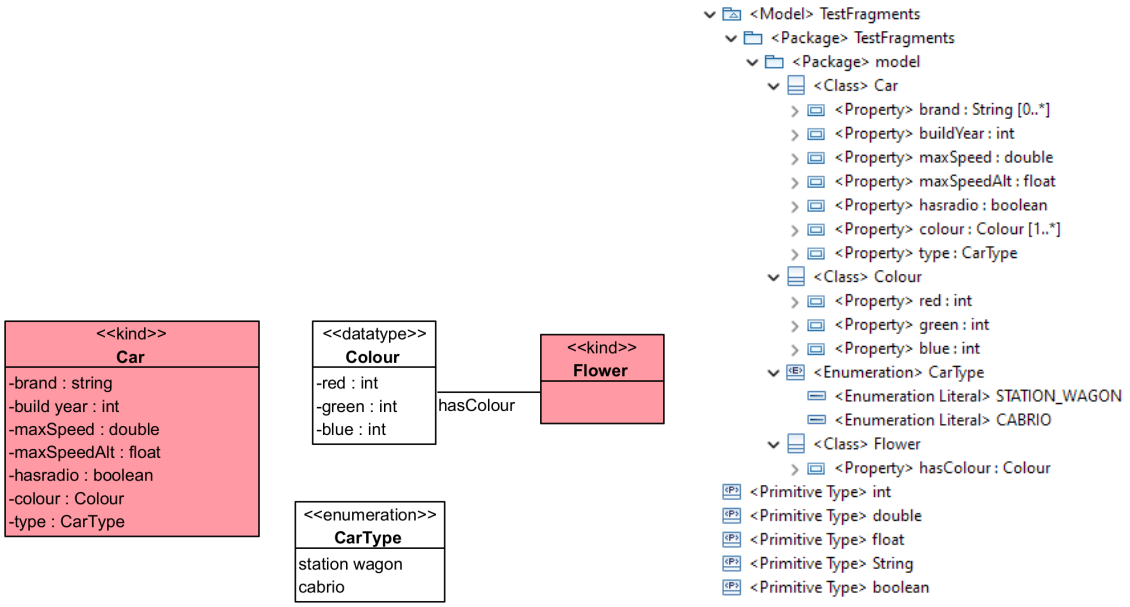
In UML, properties have an *isUnique* attribute, which indicates whether the elements of the corresponding collection are unique [9]. This information is not included in the OntoUML model, so in our transformation, we set all these attributes to *false*, i.e., we assume them to be non-unique.

The cardinalities of class properties are not displayed in Visual Paradigm, but they can be seen and altered in the specification submenu of attributes.

Relations to enumerations and datatypes Relations to datatypes and enumerations are treated differently from relations to other classes, in that they are transformed into a single property that is owned by the class pointing to the datatype/enumeration. This can be seen in Figure 5.4, where the *Flower* class has a property of type *Colour*. No association is generated in this case.

5.1.3 Other design decisions

This section describes other design decisions that influence the final result of the transformation.



(A) OntoUML fragment with class properties, an enumeration, and datatype.

(B) Generated model.

FIGURE 5.4: Transformation of class properties, enumerations, and datatypes

Naming of relation properties

The examples in the proposed transformation described in Chapter 3 often did not have names attached to relation ends. However, in practice, it is expected that some relation ends have a name in which case they should be preserved in the implementation model. This is the case for the *hasValue* property of *Flower* in Figure 5.4. When no name is present, the name used in the implementation model is derived from the type of the property. For example, if the relation from *Flower* to *Colour*is unnamed, the resulting property name would be *colour*.

Directionality of OntoUML relations

Some of the OntoUML-type transformations described in Section 3.3 rely on the directionality of relations. Consider, for example, the transformation rule of relators with mediation relations from Figure 3.13. The mediation relation has a direction from the relator to the participating role, which is indicated in Visual Paradigm with an arrowhead. When transforming this mediation relation, it is relevant to know what property belongs to the start and end of the transformation, as these two are treated differently in the transformation. In this case, the property at the end of the relation (e.g., *Employee* or *Employer* as in Figure 3.13) is marked as final.

The OntoUML metamodel contains a *PropertyStereotype* that can have the values *BEGIN* and *END*. One could assume these are used to mark the source and target ends of relations, however, they are not used as such by the VP plugin[5]. Therefore, we had to find other ways to derive the directionality of the relations for the cases in which this matters, which include *mediation*, *characationzation*, and *memberOf* relations. For this, we make

[5]The inclusion of property attributes to derive the direction of a relation would benefit the transformation developed in this research.

use of some specific behaviours of the VP plugin, which we highlight for each of these relation stereotypes.

Direction of memberOf relation In the *memberOf* transformation discussed in Section 3.3.1, we stated that the name of the property relating to the members of the collective are set to *'members'* (provided no name is given by the creator of the OntoUML model). Collectives may have other collectives as members, so we can not simply check which property refers to a collective. Rather, we assume the property marked as either aggregation or composition (visualised with either a clear or black diamond in UML) is the property relating to the owning collective.

The OntoUML VP plugin automatically marks the owning collective as an aggregation. However, this is not enforced by our Ecore model nor the OntoUML platform-independent metamodel. In case none of the properties are an aggregation or composition, both association ends are named 'member'.

Direction of mediation and characterization relations We observed that the VP plugin sets the ends of *mediation* and *characterization* relations to *isReadOnly*. This behaviour seems to align with our proposed transformation of the *charaterization* and *mediation* relations in which we mark the ends of the resulting UML Associations as *final*, as discussed in Section 3.3.4. The only difference with our proposed transformation is that for a *characterization* relation of qualities, both association ends are marked as *final* as opposed to only one of the ends.

Although this seems like a suitable solution to the *isReadOnly* attribute set by the VP plugin to derive the directionality of relations, this behaviour is not described in the OntoUML documentation nor in the literature (such as [21]). Therefore, we cannot assume this is the case for OntoUML models created with other tools, and thus, the logic used in the ATL transformation should be considered as a special design decision.

5.2 Transformation limitations

This section describes the known limitations of the currently implemented ATL transformation. The impact of these limitations on the applicability of our transformation to models of the OntoUML model catalogue is discussed in Section 6.2.3.

5.2.1 Phases in generalization sets

Our transformation currently does not support *GeneralizationSets* for phases. Generalization sets are used to represent a group of generalizations that belong together. An example of generalization sets used by phases is visualised in Figure 5.5, in which a kind *Person* has two sets of phases, one representing the position of a person relative to the ground (either *Above Ground* or *Underground*) and another one describing the life status of the person (either *Dead* or *Alive*).

For normal phase partitions, exactly one phase at a time may apply as they are classified as complete and disjoint [20, 21]. However, in the example of Figure 5.5, a person is in exactly one state of each phase partition. For example, a person can be both alive and underground, or, both be dead and above ground.

The current ATL transformation ignores generalization sets, which means that all phases belonging to another class are considered to be part of the same phase partition. E.g., for Figure 5.5 this would mean that a *Person* class is generated which can only be

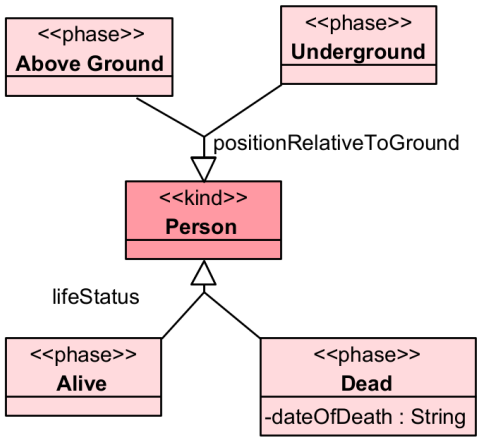


FIGURE 5.5: Example of phases in separate generalization sets.

in one phase at a time (and not, for example, both *Alive* and *Underground* at the same time).

To mitigate the impact of this limitation, the ATL transformation generates a warning message in case multiple phase generalization sets are present for a single class.

5.2.2 Multiple relations between two classes

In the OntoUML-to-implementation-model transformation, relations are transformed into bi-directional associations. Considering Figure 5.6, for example, this means that the class generated from *Legal Person* will have a reference to the class generated from *Contract* and vice versa.

Furthermore, the name of a relation end is used as the name for the generated reference. In Figure 5.6, the two mediation relations have a relation end name at only one of the ends, namely *promiser* and *promisee*. This means that the generated class for the relator *Contract* will have two references named *promiser* and *promisee*, which are both of the type *Legal Person*.

However, when such a relation end name is absent, the generated reference's name is derived from the type name. This is the case for the other relation ends of the two mediation relations in Figure 5.6, which both point towards the *Contract* relator. In the

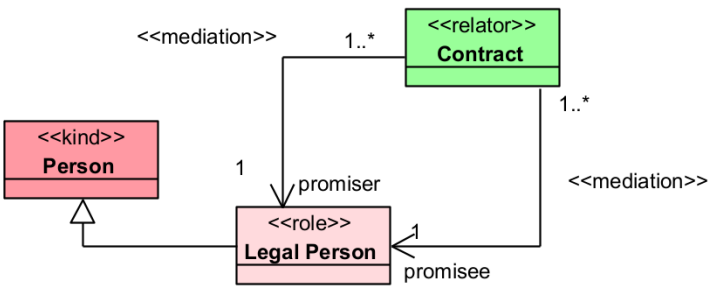


FIGURE 5.6: Example of a case in which a generated class would contain a duplicate attribute named 'contract'.

generated class for the *Legal Person* role, two references are present, which both have name *contract* (derived from the type of the relation end, namely the relator *Contract*). This results in an issue when generating code from this implementation model as class attributes should have unique names.

The example of Figure 5.6 exposes this limitation mainly in the context of the design decision to generate bi-directional associations for relation. Nevertheless, if a uni-directional association was generated (i.e., only the class generated from the *Contract* relator would have a reference to the class generated from *Legal Person* and not vice versa) this issue would not occur. However, this limitation would then still apply if the relation ends would not be given names (i.e., in case the relation ends names *promiser* and *promisee* would be omitted).

One way to work around this limitation is to explicitly define relation end names, thus letting the transformation know the appropriate reference names. Alternatively, one can avoid having multiple relations between the same two classes. For instance, in Figure 5.6, one could define a separate role for a legal person who is a promiser and a separate role for a legal person who is a promisee.

5.3 Assumptions on the source model

The transformation has been implemented assuming the provided OntoUML model is correct. By correct, we mean that it adheres to the constraints provided in [21], which can be checked by the VP plugin. This means our transformation does not check whether certain constructs are correct and what happens in these cases is undefined.

However, the constraints checked by the VP plugin are not complete. In this section, we list additional assumptions of the source model besides those verified by the VP plugin.

No properties without types In our transformation, we assume that class properties have a type. An example of properties without a valid type is displayed in Figure 5.7. The class property *age* has no type while the property *address* has a type that does not refer to either a primitive type or another existing class in the model (as discussed in Section 5.1.2).

In these cases, these properties are ignored and a warning is given on the ATL console.

No duplicate generalizations Figure 5.8 shows an OntoUML fragment in which a subkind has two attached generalizations. Subkinds are only allowed to specialize one sortal, however, the VP plugin only checks that one ultimate sortal is specialized. In this example, *Subkind2* specializes only one ultimate sortal, namely *Kind*, so no issues are raised by the VP plugin.

Although the transformation runs fine, the resulting implementation model will contain multiple inheritance, violating the constraint we impose on the implementation model

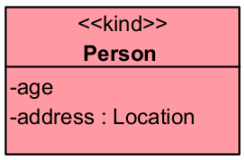


FIGURE 5.7: Example of properties with invalid types.

described in Section 3.2. There is no simple way to detect this construct as both individual generalizations are valid, hence, no warning is given by the ATL transformation.

More than one memberOf relation for a collective As discussed in Section 3.3.1, collectives may have only one type of member, however, this property is not checked by the VP plugin.

An example of a collective with multiple members is depicted in Figure 5.9. Similar to duplicate generalizations, the transformation runs fine and yields an implementation model. However, in case both *memberOf* relations do not have a predefined property name on the relation end to the member, the default name *'member'* is assigned. For multiple members, this leads to a class with duplicate names and thus results in Java code that does not compile.

5.3.1 Custom ATL warnings

The implemented transformation provides warnings in case certain constructs are detected in the OntoUML model that may not be properly handled. Three of these custom warnings are defined: one relating to the assumption that properties must have a type, one relating to the limitation regarding the lack of support for generalization sets, and one relating to empty strings present in the source model.

Contains properties without types This warning indicates that the model includes a class property without a type. As discussed in Section 5.3, these are simply ignored and thus do not result in issues for the code generation.

The warning merely serves to notify the model designer that these properties were not included in the generated code.

Multiple GeneralizationSets present for phase partition As discussed in Section 5.2, our transformation does not handle multiple generalization sets for one phase partition. In case generalization sets are present for phases, these are ignored and all phases of a specific class are considered to be part of the same phase partition.

This warning indicates that the OntoUML model contains two or more GeneralizationSets involving phases for one class. Therefore, no warning is given if only one generalization set for phases is present for a particular class, such as shown Figure 6.1, in which the

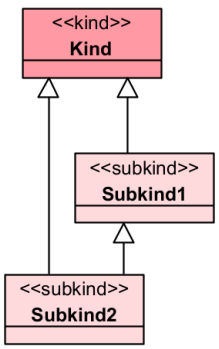


FIGURE 5.8: Example of a duplicate generalization resulting in multiple inheritance.

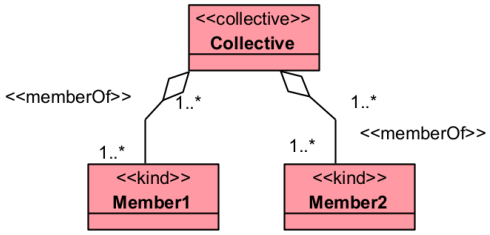


FIGURE 5.9: Example of a collective with multiple *memberOf* relations.

Planned Activity and *Performed Activity* phases belonging to the *Activity* kind are part of a single generalization set (marked as complete and disjoint).

Contains empty string The *'Contains empty string'* warning occurs in case a classifier name within the implementation model would be set to an empty string. To yield a valid Java name, non-alphanumerical characters are removed. Therefore, if a class name in the OntoUML model consists of only non-alphanumerical characters this would result in an empty string.

In these cases, the transformation gives the classifier the name *'emptystring'*. This may lead to compilation issues in case multiple classifiers in the implementation model have the name *emptystring*.

5.4 Java code generation

The final step in the transformation chain is to generate Java code from the implementation model, as illustrated in Figure 5.10. Acceleo is the model-to-text transformation tool within EMF that we used to achieve this, as described in more detail in Section 2.2.2.

As UML is a widespread modelling language, we sought to reuse an existing UML to Java code generator as opposed to implementing one from scratch. We have found such a project in the UML-Java-Generation Acceleo project[6] developed by Obeo, which is the company behind Acceleo. However, the project is not actively maintained and is marked as archived on the Eclipse project site[7]. Although the project has not been updated since seemingly 2013, it is still a comprehensive Acceleo project that addresses a lot of challenges that occur when generating code such as generating files within different packages and managing imports across these different packages.

We made some adjustments to use the Obeo UML-Java generation for our purposes. This includes setting the configuration for the transformation as well as solving some bugs.

5.4.1 Configuration of the code generation

The Obeo UML-Java generation supports a custom configuration. This configuration contains various options to tweak how Java code is generated, such as what UML packages to ignore during the code generation and whether getters and/or setters should be generated.

[6] See https://github.com/ObeoNetwork/UML-Java-Generation.

[7] The GitHub project states the UML-Java-Generation has been moved to the Eclipse UML Generators project. However, the Eclipse website states that the project has been archived. See https://www.eclipse.org/projects/archives.php.

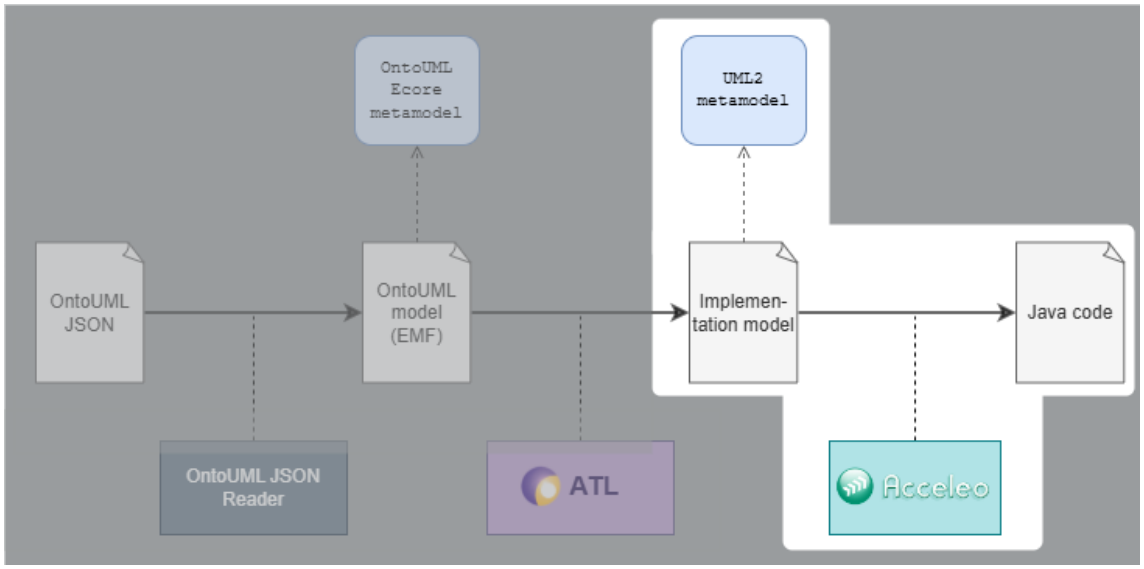


FIGURE 5.10: Transformation chain which highlights the step generating Java code from the implementation model.

TABLE 5.1: Used configuration options for the UML-Java code generation.

Configuration option	Used value
GENERATE_GETTERS_AND_SETTERS	true
GENERATE_GETTERS_COLLECTIONS	true
GENERATE_SETTERS_COLLECTIONS	true
GENERATE_ADVANCED_ACCESSORS_COLLECTIONS	true
ORDERED_NOT_UNIQUE_TYPE	java.util.ArrayList
ORDERED_UNIQUE_TYPE	java.util.LinkedHashSet
NOT_ORDERED_UNIQUE_TYPE	java.util.HashSet
NOT_ORDERED_NOT_UNIQUE_TYPE	java.util.ArrayList

The configuration options we used in our transformation are listed in Table 5.1. The first four options indicate whether getters and setters should be generated for class attributes. A special option is included for whether special methods should be added for adding and removing elements from collection attributes. In our configuration, we set these values to the most rigorous options, i.e., all possible access methods are generated[8].

The other four options describe what Java types to use for different sorts of collections. In UML, properties that have a multiplicity with an upper bound of at least one can have multiple elements in their instantiation. These properties have two attributes that describe the collection of their values: *isOrdered* indicates whether the values in a property are sequentially orderered whereas *isUnique* indicates whether or not duplicate values are allowed [9]. The UML-Java generation needs to know what Java types to use for each of the possible UML collection types. Table 5.1 shows the Java types we chose based on the Java language documentation.

Notably, $NOT\_ORDERED\_NOT\_UNIQUE\_TYPE$ is set to the Java ArrayList even though an ArrayList in Java is ordered. The reason for this is that the Java util library does not contain a non-ordered non-unique collection type. Furthermore, from a functional perspective, it does not matter if a collection is ordered when it is expected not to be ordered.

5.4.2 Fixed bugs

During the usage of the Obeo UML-Java generation Accelo project, some bugs were found that impacted the generated Java code. Each of these bugs has been resolved. The most relevant bugs for the code generation to work properly are discussed below.

Generation of attributes from association navigable owned ends

In UML, an association between two classes contains two properties, one for each end of the association. The class involved in an association may own the property referring to the other class, while the association only holds a reference to that property.

However, these properties can also be contained solely in the association. When these properties should be navigable, i.e., the generated classes should have an attribute for that property, they can be marked as *navigableOwnedEnds* in the association.

In the Obeo UML-Java project, a class attribute was generated even for association-owned ends not marked as *navigableOwnedEnd*, resulting in duplicate attributes being generated if one of the properties of the association was owned by a class. This bug was solved by adding an extra check to only generate attributes for properties included in the *navigableOwnedEnd* attribute.

Missing imports from types resulting from navigable owned ends

The Obeo UML-Java project contains an *ImportService* that derives a list of Java types that are used within a class and thus should be imported. In the original implementation, types of properties originating from navigable owned ends in related associations were ignored, resulting in missing import statements.

This issue was mitigated by adding the types of associated navigable owned ends to the list of types to be imported.

[8]The use of these most rigorous options was done to test all code generation features. An instruction on how to change these options is provided in the readme file of the project.

Not ignoring imports from the same package

In Java, classifiers located in the same package do not have to be imported. The UML-Java generation project already contained code to check whether used Java types are located in the same package and ignore these. However, this logic contained a relatively simple bug which could be fixed after which packages were ignored as expected.

Multiple interface inheritance

Java allows interfaces to extend multiple other interfaces, such as the example listed in Listing 5.4, in which an interface C extends both interfaces A and B.

The Obeo UML-Java project collected the superclasses and inserted these in the interface declaration, however, it did not add a comma between them, as shown in the lower part of Listing 5.4. We fixed this error by inserting commas between the names of the extended interfaces.

```
interface A {
    ...
}

interface B {
    ...
}

// Proper Java syntax
interface C extends A, B {
    ...
}

// Code generated with Obeo UML-Java generation bug present
interface C extends AB {
    ...
}
```

Listing 5.4: An example of multiple inheritance of interfaces in Java

5.4.3 Other design decisions

A generated implementation model might contain interfaces that have attributes or are connected to other classes through associations. This results in a challenge for the Java code generation as interfaces in Java may not contain attributes[9]. Rather, interfaces in Java may only contain constant fields or method declarations[10].

One simple solution could be to ignore UML interface attributes when generating Java code. However, this would disregard the properties of non-sortal types in an OntoUML model. Consider the example in Figure 5.11, which contains a category *Named entity* with a property *'name'*. Since the *Person* kind specializes the *Named entity* category, a person is a named entity and therefore has a name. According to the transformation rule

[9] Or fields, as they are called in the Java specification. See https://docs.oracle.com/javase/specs/jls/se21/html/jls-9.html#jls-9.2.

[10] In the Obeo UML-Java project, interface properties were transformed into constant fields (apart from navigable owned ends from connected associations). This can be considered incorrect as constant fields in Java are static i.e., attached to the classifier and not the object) and interface attributes in UML are not necessarily static.

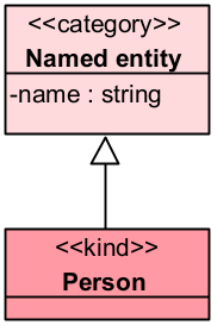


FIGURE 5.11: Example of an OntoUML model in which a category contains a property.

of categories described in Section 3.3.3, a category is transformed into a UML Interface. Furthermore, the *name* property is converted into a UML Property, as described in Section 5.1.2. Therefore, if the Java code generation ignores these properties, the resulting code loses the information that a *Person* has a name.

To illustrate our solution to this challenge, we use the sample implementation/UML model displayed in Figure 5.12. In this example, we see an interface A with property x, and an interface B which extends A and has a property y. Following the UML specification: *"If an Interface declares an attribute, this does not necessarily mean that the realizing BehavioredClassifier will necessarily have such an attribute in its implementation, but only that it will appear so to external observers"* [9]. Therefore, instead of generating an attribute in the generated Java interfaces for A and B (which is not even possible), the Java code generation defines get and set methods for these properties. The code that is generated from the implementation model in Figure 5.12 is shown in Listing 5.5, which defines a *get/setX* method in interface A, and a *get/setY* method in interface B.

The class implementing an interface should implement the methods specified by that interface. For the example in Figure 5.12, this means that class C should implement all methods specified by both interfaces A and B; Class C directly implements interface B which itself extends A and thus inherits the getter and setter for both x and y.

Listing 5.5 also shows the code generated for class C. Notably, the class does not include the fields x and y and thus the get and set methods are not generated with an implementation. The developer is free to implement the generated get/set methods in the way they prefer. Furthermore, the actual generated code includes Javadoc comments and *todo* comments highlighting to the developer that the respective methods should be implemented. In addition, the transformation takes associations connected to an interface into account, such as the association with an end named *person* belonging to the interface *PersonPhase* displayed in Figure 3.7.

5.5 Transformation chain implementation

So far, the three separate steps from the overall transformation as illustrated in Figure 5.13 have been discussed. First, an OntoUML JSON is read into an EMF-compatible format. Second, the OntoUML model is transformed into an implementation model by executing an ATL transformation. Lastly, Java code is generated from the implementation model by executing an Acceleo transformation.

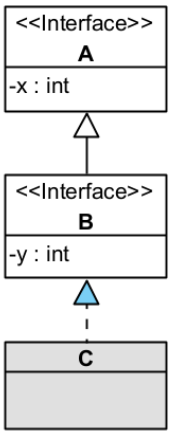


FIGURE 5.12: Example of a UML class that implements interfaces containing properties.

To generate Java code from an OntoUML model, these three steps can be executed in sequence within the Eclipse EMF environment. In addition, to facilitate the automation of running the entire transformation chain, for each of these steps, an ANT task[11] was created with the location of an input model and a location on where to store the output model (or Java code) as parameters.

Normally, an ATL transformation is executed by a run-configuration in Eclipse. Although ATL provides custom ANT tasks to execute a transformation, these tasks all depend on the Eclipse runtime environment. To allow the execution of the transformation in other contexts, we want to be able to execute these ANT tasks from outside of Eclipse. Therefore, we exported our ATL project to an Eclipse ATL plugin which includes a Java adapter for the execution of the transformation, hence allowing the transformation to be run as a standalone application.

[11] See https://ant.apache.org/.

```java
public interface A {
    int getX();
    void setX(int newX);
}

public interface B extends A {
    int getY();
    void setY(int newY);
}

public class C implements B {
    @Override
    public int getX() {
        return 0;
    }

    @Override
    public void setX(int newX) {}

    @Override
    public int getY() {
        return 0;
    }

    @Override
    public void setY(int newY) {}
}
```

Listing 5.5: Generated Java code for UML interfaces with properties and a class that implements these interfaces.

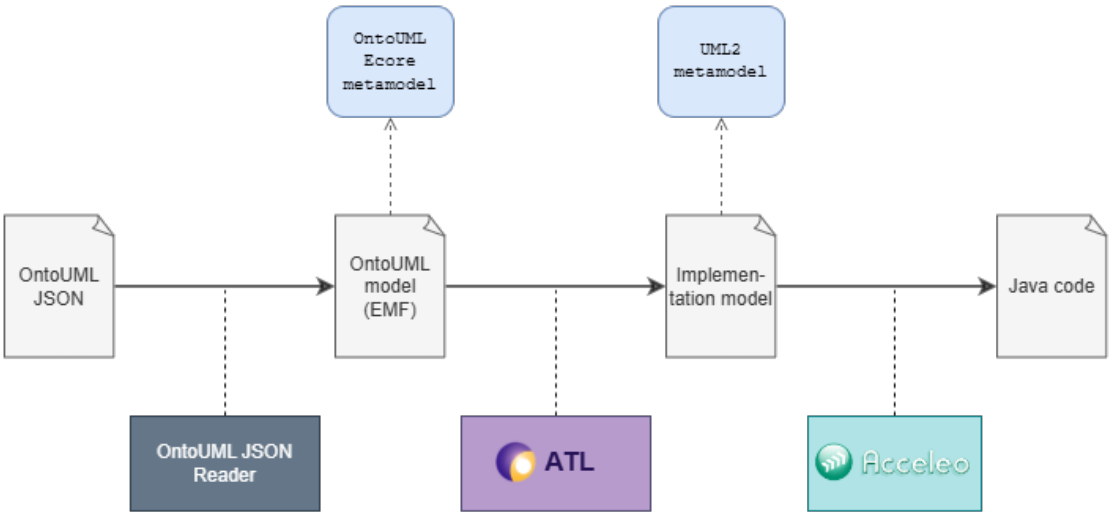


Figure 5.13: OntoUML-to-Java transformation chain.

Chapter 6

Validation

This chapter describes the validation of our OntoUML-to-Java transformation. Section 6.1 illustrates how a model is propagated through the transformation chain using the example of the Project Management Ontology. Section 6.2 describes the automated validation process, in which 82 models from the OntoUML model catalogue have been taken as input to the OntoUML-to-Java transformation. The generated code was checked for compilation and for the code that did not compile the cause was analysed. Section 6.3 describes the manual validation process of five models in which the OntoUML model was compared with the generated Java code to check whether the expected code patterns were generated.

6.1 Full transformation example

This section shows the different steps in the Java code generation for the Project Management Ontology, which is one of the models from the OntoUML model catalogue[1]. This ontology, which is displayed in Figure 6.1, was chosen because it contains a variety of OntoUML Class stereotypes that are all supported by our transformation, and is neither too small nor too large for illustration purposes.

OntoUML EMF model The first step of the transformation is to read the OntoUML JSON file from the Project Management ontology into an EMF-compatible format. This EMF model is an instance of the OntoUML metamodel described in Section 4.1.

Generated implementation model Figure 6.2a displays the implementation model generated from the OntoUML EMF model. Figure 6.2a shows all the UML Classes and other elements generated for the Project Management ontology. Two interfaces are generated, one for the roleMixin *Resource*, and one *ActivityPhase*, the latter originates from the complex phase partition of the *Activity* kind. Figure 6.2a does not show that both classes *PlannedActivity* and *PerformedActivity* have an *InterfaceRealization* towards the *ActivityPhase* interface. Similar to the *InterfaceRealization*, not all the *Generalizations* are directly visible in Figure 6.2a

Generated Java code The final step in the transformation chain is the generation of Java code from the implementation model. Figure 6.2b displays the Java files generated

[1] See https://github.com/OntoUML/ontouml-models/tree/master/models/project-management-ontology.

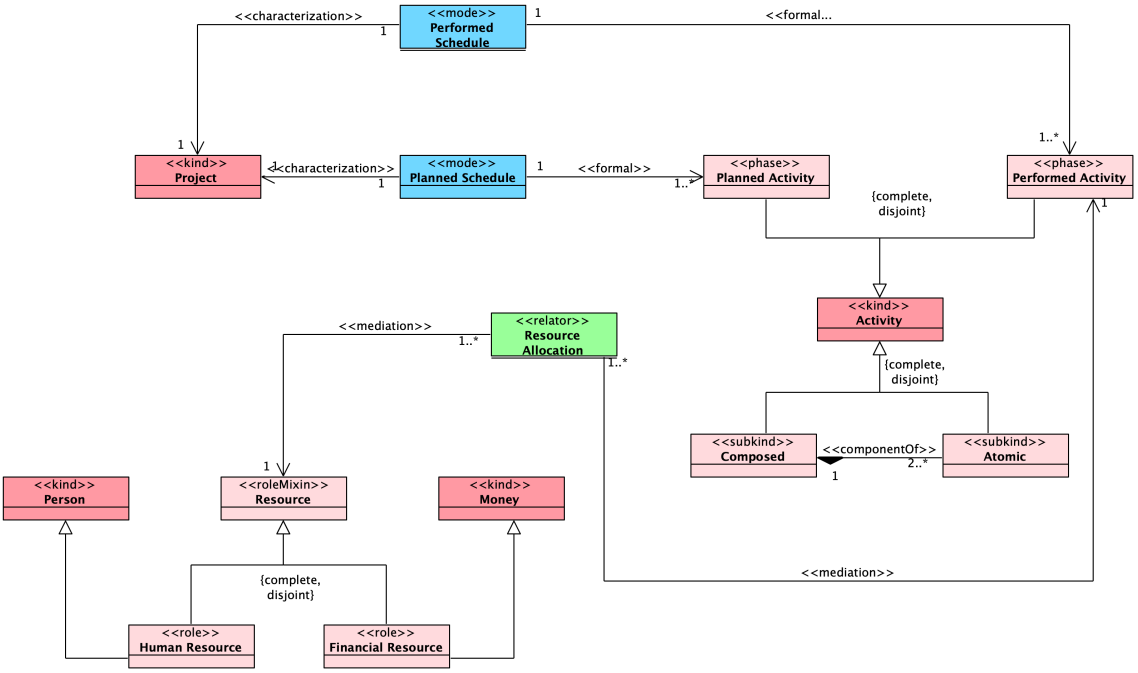
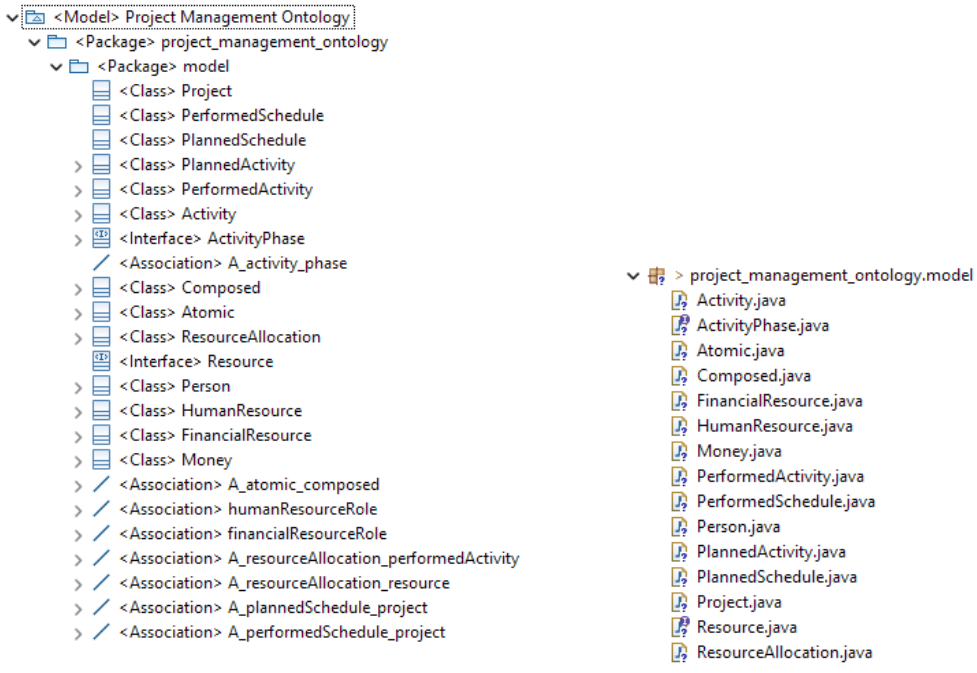


FIGURE 6.1: Project Management Ontology.



(A) Implementation model. (B) Java files.

FIGURE 6.2: Results of the transformation chain for the Project Management ontology.

from the implementation model, which correspond to the classes and interfaces showed in Figure 6.2a.

Two generated Java files are highlighted to illustrate the generated code. In both of these examples, the generated Javadoc strings are omitted to keep the code examples short[2].

Listing 6.1 lists the Java code generated for the subkind *Composed*. In the code, we see that the class extends the *Activity* class (corresponding to the *Activity* kind) and has an *atomic* attribute corresponding to *componentOf* relation referencing the *Atomic* subkind. The type of this attribute is an *ArrayList*, as it has a multiplicity of possibly many and is marked as unordered and not unique in the OntoUML model[3].

Listing 6.2 lists the generated Java code for the *Financial Resource* role. *Financial Resource* class implements the *Resource* interface, which corresponds to the respective *Resource* roleMixin. Furthermore, the *money* attribute which refers to the kind of this role is also generated. Following the transformation of roles as discussed in Section 3.3.2, this property is marked as final and only a get method is generated.

```
1  package project_management_ontology.model;
2
3  import java.util.HashSet;
4
5
6  public class Composed extends Activity {
7
8      public ArrayList<Atomic> atomic = new ArrayList<Atomic>();
9
10     public Composed() {
11         // Start of user code constructor for Composed)
12         super();
13         // End of user code
14     }
15
16     // Getters and setters
17     ...
18
19 }
```

LISTING 6.1: Java code generated for the *Composed* subkind class from the Project Management Ontology.

6.2 Transformation of OntoUML models from the catalogue

In an additional validation effort, we have executed the OntoUML-to-Java transformation on models from the OntoUML model catalogue. The goal of this effort was to identify bugs and assess the feasibility of the transformation. As such, for each of the transformations, we were interested in whether the transformation was successful, the execution time of the transformation, and whether the generated Java code could be compiled.

[2]The complete version of the generated code can be found on https://github.com/GuusVink/ontouml-java-generation.

[3]The *unordered* and *not unique* properties are contained in the OntoUML JSON but can not be visually derived from the diagram in Figure 6.1.

```java
package project_management_ontology.model;

public class FinancialResource implements Resource {

    public final Money money = null;

    public FinancialResource() {
        // Start of user code constructor for FinancialResource)
        super();
        // End of user code
    }

    public Money getMoney() {
        return this.money;
    }

}
```

LISTING 6.2: Java code generated for the *Financial Resource* role class from the Project Management Ontology.

6.2.1 Methodology

We developed a Python script to execute each of the ANT tasks described in Section 5.5. Besides the steps of the transformation, we defined a separate ANT task to compile the generated Java code with the latest LTS version of Java, namely SE 21.

For each analysed model, data about the different transformation steps was collected, such as the time the transformation step took and the warnings given by the ATL and Acceleo transformation. In case the transformation failed, the step at which it failed was collected as well as the error log. In the final step, the ANT build task on the generated code was executed to check whether the code could be compiled, and again, in case the build failed the error log was recorded.

Generated Java code that compiles is not a guarantee that the transformation was faultless. To check whether the transformation worked entirely as intended, one would have to go over the OntoUML model to check whether the generated code corresponds to the expected constructs of the implementation model as presented in Chapter 3[4]. Still, compilable generated code at least indicates that a valid implementation model was generated.

This validation process was performed on the 82 models from the OntoUML model catalogue mentioned in Section 3.1.3 that only contain class stereotypes covered by our transformation. The technical specifications of the processing environment used to execute the transformation can be found in Section B.1.

6.2.2 Results

The results of the validation process are given in Appendix B. A summary of the results is provided in Table 6.1.

During the validation, 14 models were found that included the invalid relation stereotype *'Formal'*, which is a deviation from the valid *'formal'* stereotype. Up until 2024, the *formal* stereotype was not supported by the VP plugin, even though it has been consid-

[4]Manual validation was also performed on some models, which is described in Section 6.3.

ered a valid stereotype[5]. Presumably, users of the VP plugin have worked around this by adding a custom stereotype (sometimes spelled as 'Formal'). Since the *formal* stereotype is currently a valid OntoUM stereotype, we have decided to rename all occurrences of *Formal* with *formal* to make these models compatible with the current OntoUML model, as opposed to disregarding these 14 models.

The summary of the validation results in Table 6.1 shows that 79 out of the 82 models were successfully transformed, meaning that some Java code was generated. The 3 models that failed did that at the step of reading the OntoUML model, as these contained stereotypes considered invalid by the metamodel. From the 79 models that yielded Java code, 55 resulted in compilable code (69.6%).

The average execution time of the entire transformation chain was 16.54 seconds, of which most time went to both the ATL and Acceleo steps (both around 8 seconds). The execution time for the ATL transformation from the ANT task (as discussed in Section 5.5) deviates quite a lot from the execution time within ATL: whereas the ATL ANT task has an observed time of approximately 8 seconds, the transformation only takes around 1 second when executed in Eclipse. A possible explanation of this could be that the ATL plugin requires a startup phase that has to be executed each time the ANT task is executed from outside of Eclipse. In contrast, when running the transformation from the Eclipse environment, this startup phase could be contained in the startup process of Eclipse itself.

6.2.3 Analysis of the fault modes

Table 6.2 provides a summary of the occurrences of ATL warnings as well as different categories of compile errors that occurred in the transformations.

First, we will go over the different ATL warnings and consider their implication. Afterwards, we will analyse the compile errors that occurred and determine whether these arise due to issues in the transformations or because the OntoUML model contains constructs we consider invalid.

ATL Warnings

The first part of Table 6.2 shows the occurrence of all the ATL warnings described in Section 5.3.1. The *Property without type* warning is the most frequent with 25 occurrences. As

[5]This issue was discussed in the GitHub project of the VP plugin, as can be found on https://github.com/OntoUML/ontouml-vp-plugin/issues/84.

TABLE 6.1: Results of validation performed on the models from the OntoUML model catalogue.

Metric	Value
Number of models analysed	82
Number of successful transformations	79
Number of projects resulting in compilable code	55
Average time per successful transformation	16.54s
Average time per transformation step	
Read OntoUML model	1.02s
ATL transformation	7.48s
Acceleo generation	8.04s

TABLE 6.2: ATL warnings and compilation errors for models from the OntoUML model catalogue.

ATL Warning	Occurrence in # models
Contains property without type	25
Multiple GeneralizationSets present for phase partition	1
Contains an empty string	1
Compile error category	
Duplicate variable definition	16
Reserved keyword used as name	4
Multiple class inheritance	3
Invalid override from superclass	1

properties without types are simply ignored, there are little consequences to the generated code.

One of the models yields the *multiple-phase generalization sets* warning, namely the *music-ontology*. Therefore, we conclude that for now, the impact of this limitation is minimal.

Finally, there is one model that yields a *empty string* warning. After a manual inspection, this warning originates from the *fraller2019abc* model. This model contains two classes named *'...'*, which both yield an empty string after removing the invalid *'.'* characters.

Categories of compile errors

In case the compilation of the generated Java code fails, the error logs are collected. From these error logs, four categories of compile errors have been found, which are listed in Table 6.2 along with the frequency of their occurrence.

Duplicate variable definition The 'Duplicate variable definition' error occurs when a classifier contains two or more attributes and/or methods that have the same name/method signature. This error occurred for the code generated from 16 projects.

Six of these models contain a duplicate variable *members* as they contain collectives with more than one type of member, which we consider as invalid OntoUML as discussed in Section 5.3.

Two other models, namely *junior2018o4c* and *fraller2019abc*, have multiple relations defined between the same two classes, without uniquely named relation ends. This corresponds to the limitation of the transformation described in Section 5.2.2.

The model *jacobs2022sdpontology* yields code that contains a duplicate variable definition which can not be directly related to the source OntoUML model. Figure 6.3 displays a fragment of this model, which shows a *Business Role* role, which is a specialization of the *Role* kind, and contains a property named *role*. Following the transformation of roles described in Section 3.3.2, the generalization is transformed into a UML association that derives the names of the association ends from the names of the involved classes.

In the case of Figure 6.3, this means that the class generated from *Business Role* contains a reference to the class generated from the *Role* kind named *'role'*. However, the *Business Role* class already contains a property with that name. Although the transformation works as expected, this somewhat unfortunate combination of model elements

yields code that cannot be compiled. We see no simple automated solution to mitigate this issue and therefore suggest changing the names of properties in the source model if such issues occur.

The causes of the occurrence of the 'duplicate variable definition' error in the other seven models are discussed in more detail in Section B.3. In all these cases, we identified an issue in the OntoUML model that caused errors in the generated code.

Reserved keyword used as name The compilation errors in the category 'Reserved keyword used as name' arise when a model element in the OntoUML model contains a name that corresponds to a Java reserved keyword.

This is the case for four of the analysed models. Mdel *demori2023miscon* contains a *kind* with name *Interface*, while models *srro-ontology*, *university-ontology*, and *zanetti2019orm-o* contain a *subkind*, *role*, and *kind* named *Class*, respectively.

Multiple class inheritance The 'Multiple class inheritance' error occurs when the generated code contains a class that extends multiple other classes. This error occurs in the code generated from three OntoUML models[6].

After manual inspection, we realised all these models have a *subkind* that specializes two or more other classes. As discussed in Section 5.3, we consider these models to be invalid.

Invalid override from superclass The 'Invalid override from superclass' category occurs when a subclass contains a method with the same signature as the superclass, but has a return type that is not a subtype from the superclass method's return type[7].

This error occurred for the *music-ontology*, of which a fragment is displayed in Figure 6.4. This model has a *Group* collective with two successive subkinds, namely *Artistic Group* and *Musical Group*. All these three collectives have a *memberOf* relation to other classes, which yield, among other methods, three *getMembers()* methods, each overriding the respective method in the superclass. However, these three methods have different return types, namely *'ArrayList<Member>'*, *'ArrayList<ArtisticMember>'*, *'ArrayList<MusicalMember>'*[8].

OntoUML does not restrict the usage of redefined properties but also does not clearly support it. To illustrate this, although the OntoUML JSON contains an attribute *redefinedProperties* for the *Property* model element (as shown in Figure 4.3), this attribute seems always to be set to null and thus it seems that this feature is at least not supported by the VP plugin. Therefore, we did not address this in our implemented transformation.

[6]The 'Multiple class inheritance' error occurs for the OntoUML models 'aguiar2019ooco', 'buridan-ontology2021', and 'digitaldoctor2022'.

[7]See https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.4.8.

[8]Although all of these types belong to the same hierarchy, i.e., *MusicalMember* extends *ArtisticMember* extends *Member*, they all are encapsulated in lists, which are invariant in Java.

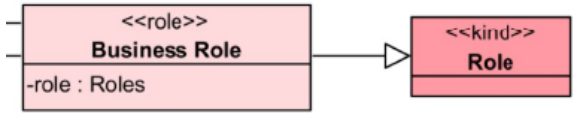


FIGURE 6.3: Fragment of the *jacobs2022* model.

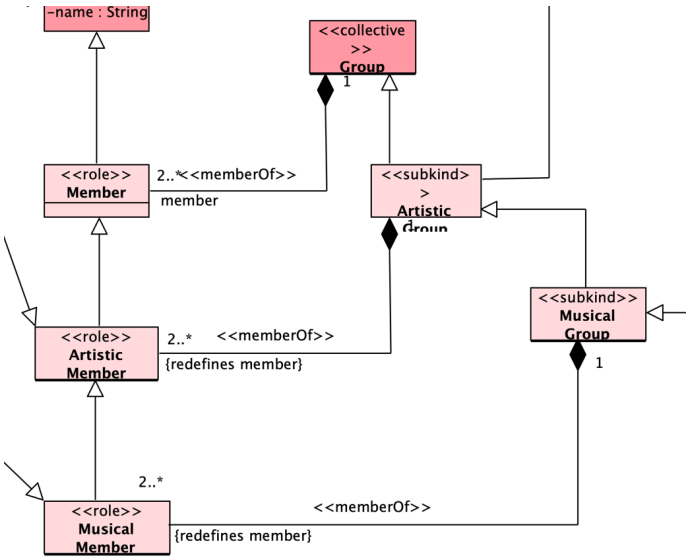


FIGURE 6.4: Fragment of the *music-ontology* which contains a UML redefines property string.

We deem the impact of this limitation rather limited for now as it only applies to one of the analysed models.

6.2.4 Performance analysis

To estimate the scalability of the transformation, we evaluated how the execution time of the transformation differs for different sizes of OntoUML models. As stated before, the execution time was recorded when performing the transformation on models from the OntoUML model catalogue. Next to this, we take the number of classes as an indicator of the size of an OntoUML model[9].

Figure 6.5 plots the number of classes against the execution time, where each point corresponds to one OntoUML model. In this plot, only the 79 models that successfully went through the entire transformation chain are included as the unsuccessful ones terminated prematurely and thus have a shorter execution time. Furthermore, we did not include the model *barcelos2015transport-networks* in this plot as it consists of 487 classes and would disturb the diagram. Figure 6.5 also includes a grey dashed line which represents the datapoints under a linear assumption[10].

The distribution of the data points shows that there are around 16 seconds of base time for the entire transformation chain. This time can be attributed to the startup times of both Java, ATL and Acceleo. Although there seems to be a trend of an increased transformation time for larger models, the correlation seems weak. When assuming this trend holds, the slope of the fitted line indicates that the execution time increases with approximately 0.14 seconds for every 10 classes in an OntoUML model.

The execution time of the transformation for the *barcelos2015transport-networks* model is 20.99 seconds. When using a time increase of 0.14 seconds per 10 classes and a base

[9]Although classes roughly indicate the size of a model, other model elements such as relations and generalizations presumably also impact the transformation execution time. Furthermore, two models with the same number of classes may contain a different number of other model elements.

[10]This line was calculated using the Numpy polyfit function, see https://numpy.org/doc/2.0/reference/generated/numpy.polyfit.html.

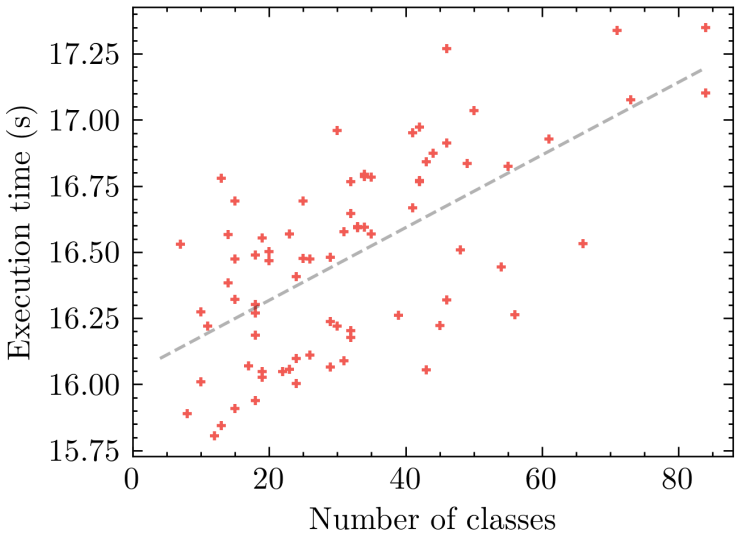


FIGURE 6.5: Transformation execution time against the number of classes of the OntoUML models.

time of 16 seconds, the expected execution time of the *barcelos2015transport-networks* would be 22.82 seconds[11], which slightly overestimates the measured transformation time. Therefore, this linear trend of execution time for the number of classes still seems to hold for larger models.

6.3 Manual validation

The automated validation described in Section 6.2 has only checked whether the OntoUML-to-implementation-model transformation runs without throwing errors and whether the generated Java code can be compiled. Although this gives some indication of whether bugs are present, it does not ensure the correctness of the transformation. For example, it is possible that the generated implementation model is not complete, i.e., it misses expected elements, but can still be compiled.

To increase confidence in the correctness of the implemented transformation, a manual validation step has been performed on five of the models in the OntoUML model catalogue, namely *aguiar2018rdbs-o*, *bank-account2013*, *bank-model*, *barcelos2013normative-acts*, and *barros2020programming*. These are the first five models (in alphabetical order) that yield compilable code[12].

The checklist provided in Appendix C has been used to verify the generated code of each of the selected OntoUML models. In case all checklist items pass, we consider the model validated.

In addition to these models from the model catalogue, the transformation has also been tested with smaller custom-made fragments of OntoUML, representing both the source models used in Section 3.3 as well as other edge cases. These fragments were mainly used during the implementation and debugging of the ATL transformation[13].

[11] Calculated by: $487 \cdot 0.014 + 16 = 22.82$.

[12] Also, the model *barcelos2015transport-networks* has been skipped in this selection due to it consisting of 487 classes, which would be tedious to check manually.

[13] The JSON files and screenshots of these fragments are included in the repository at

The filled-in checklists for manual validation of the models are included in Appendix C; all five models passed this checklist. Although the transformation worked as intended, we encountered some issues with the analysed OntoUML models. Furthermore, we list some things we encountered that inspired us to adjust our transformation. Both are discussed below.

Invalid OntoUML constructs Three of the manually analysed models contained constructs that would be marked as invalid by the OntoUML VP plugin. *aguiar2018rdbs-o* contains subkinds that do not specialize any identity provider. Furthermore, *aguiar2018rdbs-o*, *bank-account2013*, and *bank-model* contain relators that specialize other relators, which is invalid as ultimate sortals may not specialize other ultimate sortals[14].

Changes made to the transformation Based on the manual validation, three cases inspired us to adjust the implementation of the transformation:

- *barcelos2013normative-acts* contained class properties with the type *'char'* (as in a single character). In the original implementation, *char* was not supported as a valid primitive type, as defined in Section 5.1.2, meaning that they were originally ignored. In the current version, *char* is included as a valid primitive type for properties.
- *bank-account2013* contains an OntoUML role that extends another role. In the original implementation, only roles that specialized rigid types were supported. Based on the online OntoUML documentation, we decided that roles specializing other roles should be supported, which was successfully adjusted in the ATL transformation.
- *bank-account2013* and *barros2020programming* contain Portuguese names for model elements. In the original implementation, all non-alphanumerical non-ASCII characters were removed to yield valid Java names. This meant that special letters in models would be removed in the implementation model. For example, 'Operação' would become 'Operao'. Based on the use of the Portuguese language in some models and Java's support for special letters, an adjustment was made to support these characters. The implementation of this feature is described in Section 5.1.1.

https://github.com/GuusVink/ontouml-java-generation.

[14]The proper way to model this would be to use subkinds of relators.

Chapter 7

Related work

In an earlier stage of this work, we performed literature research to identify existing methods for the application of Ontology-Driven Software Development (ODSD), which is an extension of MDE where ontologies take the place of models [30, 22]. From the identified approaches, three specifically relate to the use of OntoUML[1], where one describes the use of OntoUML as a reference ontology and two use OntoUML to automatically generate system components. The latter two are considered related to this work. Furthermore, one other approach that is described was identified after this research was started.

These approaches all aim to preserve the semantics of an OntoUML model in components of an application. However, they differ in scope and overall goal compared to the work presented in this research. Still, they have some overlap in the methods used to preserve OntoUML semantics as well as their applicability in the development of software applications based on OntoUML.

Section 7.1 describes the generation of relational schemas for databases from OntoUML models [13, 14]. Section 7.2 describes the transformation of OntoUML into an information model, which is mainly focused towards separating ontological concerns from informational concerns [4, 3]. Section 7.3 describes the transformation of OntoUML into an implementation model on a conceptual level [32].

7.1 Generation of relational schemas

A relational schema describes the structure of a relational database. The primary constructs in schemas are tables, which consist of columns (features or references to other tables) and rows (individual data entries). These schemas can be automatically generated based on a conceptual model [13]. The transformation of a conceptual model to a relational schema is not trivial, as conceptual modelling languages can contain certain constructs that are not directly supported by relational schemas.

As discussed in [13, 14], multiple transformations have previously been defined to generate a relational schema from a conceptual model. Each of these transformations maps certain classes from the conceptual model to tables within a relational schema. The most straightforward approach is *one table per class*, in which each class yields a single table. However, this results in many tables, which might be undesirable in a database. Other transformations merge multiple classes into a single table, such as the *one table per leaf class* and *one table per hierarchy* transformations.

In [13], a strategy is proposed that is based on OntoUML, namely *one table per kind* [13].

[1] Five other identified ODSD approaches use the OWL language [24].

As the name suggests, this transformation generates one table for each OntoUML kind, and attributes and associations from sub- and superclasses are merged into the table representing the kind. Kinds are considered the fundamental elements within a domain [14] so that the resulting relational schema is expected to be more understandable from a human perspective. Furthermore, as kinds are less likely to change over time, the relational schema is hopefully better maintainable in case of future changes in the conceptual model.

Although the relational schema is based on the conceptual model, it still contains less information than the original conceptual model. The research identifies the constraints that are lost in the *one table per ** transformations [14]. These missing constraints originate from the two operations used in these transformations, namely flattening and lifting. The lost semantics is specific to the relational schemas and thus apply to all transformations and not only to the ones specifically based on OntoUML.

To mitigate the lost semantics, [14] provides an approach that generates database triggers. These database triggers detect situations in the database that violate constraints present in the source model that cannot be represented in the relational schema.

[14] states that the generation of relational schemas from OntoUML models is supported by the OntoUML VP-plugin. However, at the moment of writing, this feature is only implemented in a separate branch of the project, and is not included in the default installation of the OntoUML VP-plugin.

Limitations

The generation of relational schemas as described in [13, 14] enables the creation of a database for an application in the domain of a certain OntoUML model. In this scenario, an application needs to be created to access and work with the data in the database. Provided that the application is written in an object-oriented language, an object model is required. One could consider creating a class for each table in the database, however, these classes do not reflect the database triggers that describe the semantics.

Furthermore, as discussed in, for example, the transformation of roles in Section 3.3.2, we can argue that a separate class should be created for each role that contains all functionality related to that role. This would not be the case when a class is generated for a table in the database resulting from the approach described in [13], as the properties of the role are lifted to the kind, and thus are merged into a single table.

7.2 Generation of information model

In [3, 4], the authors mention that OntoUML ontologies are often designed with certain modelling goals that are not necessarily useful for information systems. Therefore, they aimed to separate these ontological and informational concerns. This has been achieved by generating an information model from OntoUML that is a technology-independent representation in UML. The addressed informational concerns are split into two categories: informational demand, which addresses pragmatic requirements such as history tracking and measurement; and representation, which addresses the usage of a modelling language to correctly organize elements.

The representational concerns relate to the choices made in transforming an OntoUML model into an information model and therefore relate to the transformation described in Section 3.3. The transformation choices made in [3] are discussed below.

Transformation of kinds, subkinds, and categories

Each kind, subkind, and category from the OntoUML model is transformed into a UML class, as shown in Figure 7.1. The OntoUML stereotypes are omitted in the resulting model.

Transformation of roles

The transformation of roles is a bit more complex. In the final transformation, each role is not transformed into a class, but it is mapped to a property of the class representing a kind to which the role belongs in the OntoUML model.

To illustrate this, consider the example in Figure 7.2. The *Husband* role is transformed into a property that points to the class representing the *Mariage* relator. Relators are individuals that connect entities [21], such as the marriage displayed in Figure 3.5, which connects a husband to a wife. Due to this decision, the transformation relies on a relator to be present in the OntoUML model.

Transformation of roleMixins

In [3], a transformation for roleMixins is also defined. Figure 7.3 shows an example of a roleMixin, namely the *Customer*. A customer can be two types, either a *Private Customer* which is a *Person*, or a *Corporate Customer* which is a *Organization*. The *Supply Contract* relator relates any supplier (necessarily an organization) to a customer (either a person or organization).

The roleMixin is transformed into a super-type for both the classes that represent the kinds that have roles specializing this roleMixin. The rest of the transformation is the same as for normal roles, i.e., a class is generated for the Relator and the role names are included as association names.

Limitations

The proposed transformation from OntoUML to an information model in [3] is implemented within EMF. However, the proposed approach has some limitations. Firstly, only a subset of the OntoUML types is covered. The extension of this transformation to other types is mentioned as a topic for future work. Secondly, the EMF Ecore model that is used is a custom OntoUML representation, which is not compatible with the implementation-independent OntoUML metamodel described in Section 2.1.3. This means that this metamodel is incompatible with models created with the OntoUML VP plugin.

7.3 From domain ontology to implementation model

In [32], the authors aimed to generate an implementation model from OntoUML models. In this work, an implementation model is considered an object-oriented model that can be trivially transformed into code in any object-oriented programming language. Although they do not mention any specific modelling language, apparently they use UML Class diagrams to represent implementation models. As an added constraint for this implementation model, they state that it should not exhibit multiple inheritance, which is stated as being a complex construct that is not supported by most programming languages.

In their work, they describe a transformation of OntoUML to this implementation model on a conceptual level. The transformation can be characterized in three distinct parts:

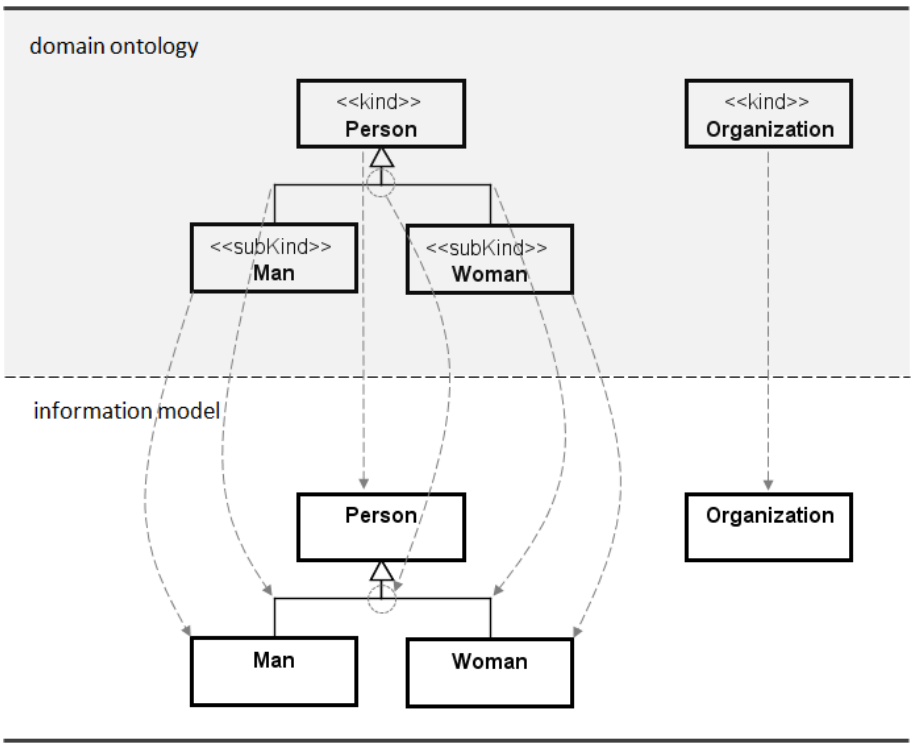


FIGURE 7.1: Transformation of kinds, subkinds, and categories to an information model in [3].

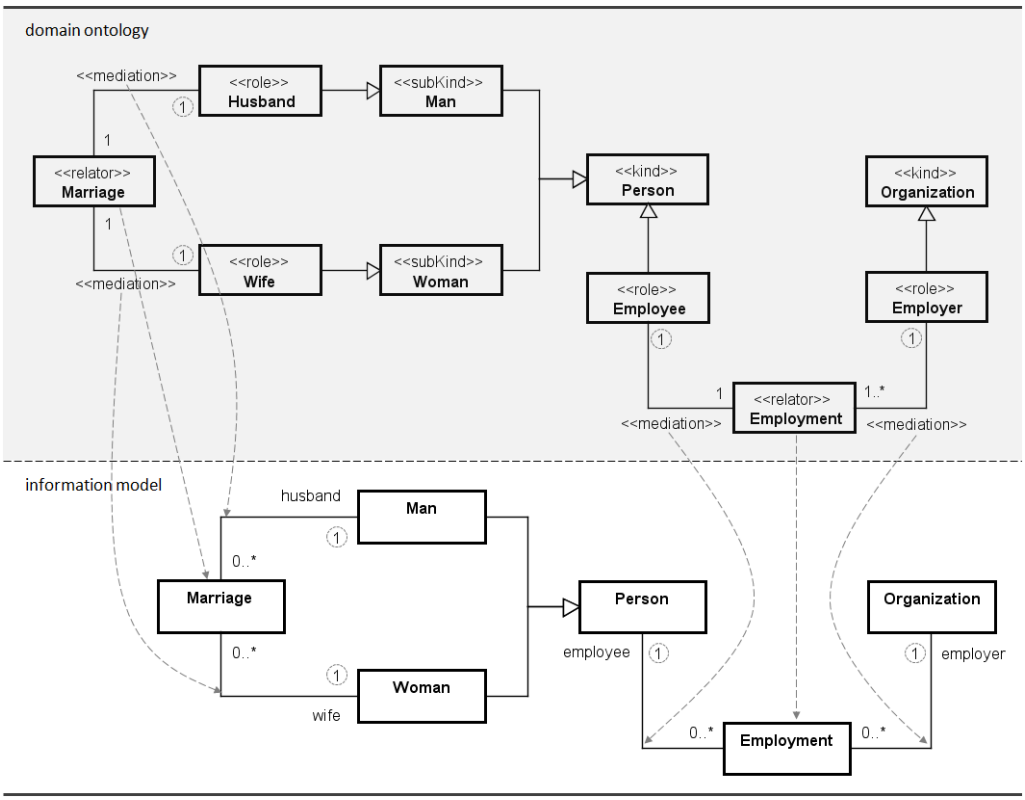


FIGURE 7.2: Transformation of roles with a relator to an information model in [3].

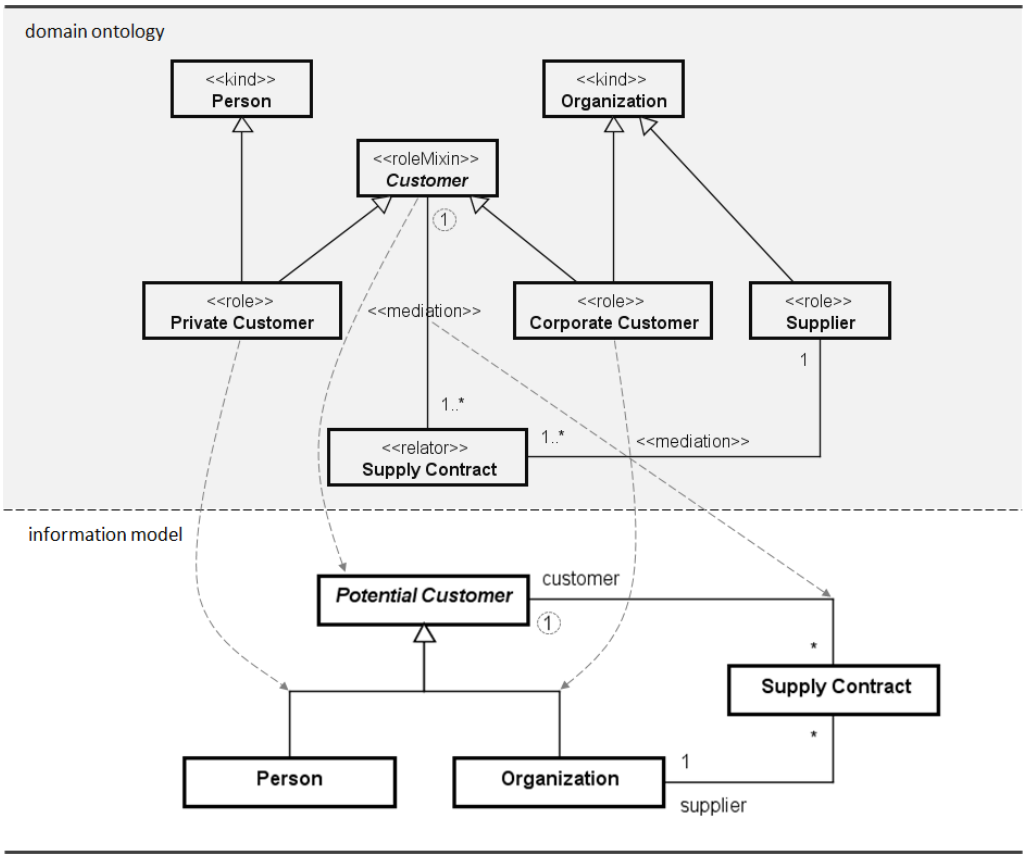


FIGURE 7.3: Transformation of roleMixins to an information model in [3].

1. Transformation of attributes and associations along with their multiplicities.
2. Separation of state and identity.
3. Transformation of the OntoUML specific types.

Attributes and associations

As OntoUML is defined as a UML class diagram extension, the only difference with UML is that OntoUML uses UFO-specific stereotypes, where the classes, attributes, associations, and their multiplicities have the same meaning as in plain UML. Thus, the transformation of these constructs to the object model is the same as for plain UML and therefore considered trivial.

Separation of state and identity

To address the differences in the rigidity of certain UFO types, a separation is made between the state and identity of an object. The notion of state and identity is taken from Clojure, which is a functional programming language defined in the Java platform[2]. Identity focuses on the non-changing part of an object, while the dynamic characteristics are captured in its state. Thus, each class in the source OntoUML model is mapped to two classes in the implementation model, for state and identity, respectively.

From an initial viewpoint, this separation of identity and state might seem like an unnecessary complexity. However, it proves useful for some UFO types because of the differences in rigidity. For instance, anti-rigid types such as roles are attached to a specific state, so, when a role changes, the state of an object can change while maintaining the history of all previous states of that object.

Transformation of OntoUML specific types

Only the transformation of kinds, subkinds, roles, and phases is described in [32]. Figure 7.4 shows a sample transformation of an OntoUML ontology to an implementation model. The *Person* kind is transformed into an identity that can have multiple states over time, which is represented by an ordered set. The *House* kind is also transformed into a state and identity, however, Figure 7.4 leaves this out to keep the diagram simple.

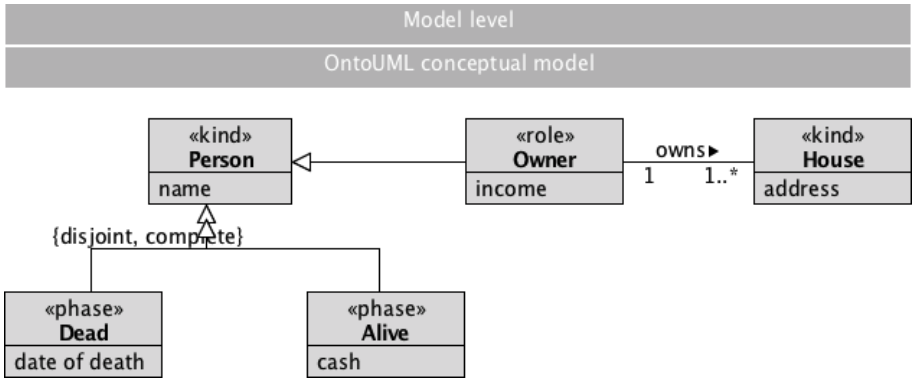
The transformation of a role is less straightforward. In OntoUML, roles are specialisations of kinds. In implementation models, specialisations also exist. However, in UML and object-oriented languages, specialisations are rigid constructs whereas in OntoUML they are anti-rigid. Therefore, the specialisation is transformed into an association attached to the state of a kind, which allows the role to be updated during the lifetime of a kind.

Finally, the phases are transformed into UML classes that are also attached to the state. Similar to roles, the specialisation is transformed into an association. Furthermore, these associations to the phases are restricted by an XOR constraint. This is a UML feature that indicates that the class (*PersonState* in this case) may only have one of the associations set.

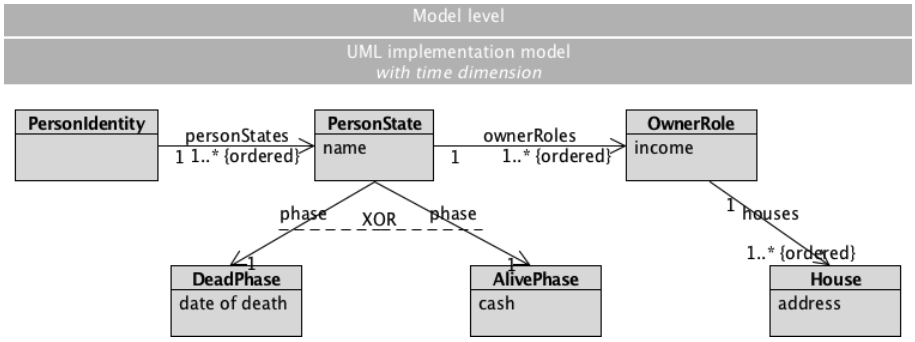
Limitations

The approach presented in [32] which has not been implemented. Some challenges stated as future work are to ensure the correctness of the transformation for different programming

[2]See https://clojure.org/.



(A) OntoUML model with a kind, role, and phases.



(B) Resulting implementation model after the transformation.

FIGURE 7.4: Sample transformation from an OntoUML ontology to an implementation model. (Copied from [32])

paradigms/languages, the transformation of other OntoUML types, and the availability of user-friendly tools to work with these transformations. These are mentioned as prerequisites for the transformation to be used in practice.

Furthermore, although the approach described in Section 7.3 separates classes in an identity and state, we opted to not do this for our transformation. We think that separating one concept into two classes deviates from what is considered normal practice for most programmers, thus raising the effort required to understand the structure of the generated code and the ability to start implementing functionality. However, it is possible that separating state and identity in different classes yields better code, which should be kept in mind when performing case studies with our proposed transformation in future work.

Chapter 8

Discussion

This chapter discusses some of our findings as well as possible ways the implemented transformation can be extended. Section 8.1 discusses the discrepancies between different OntoUML tools and how a change of the OntoUML JSON file exported by the VP plugin affects the developed transformation. Section 8.2 discusses the limitations of the performed validation and suggests how this could be improved in the future for general ATL transformations. Section 8.3 discusses the possibilities of using the implementation model to generate code for other object-oriented languages. Section 8.4 discusses an alternative to the implementation model using a UML profile and how this could be used to adapt the definition of the implementation model if desired. Finally, Section 8.5 discusses possibilities of how data persistence could be implemented in the generated code, also relating to the method to generate relational schemas with OntoUML described in Section 7.1.

8.1 Alignment of OntoUML tools

In this research, we developed an OntoUML EMF metamodel based on the JSON output of the VP plugin. However, the OntoUML ecosystem also includes the OntoUML JSON Schema and implementation-independent metamodel, which in theory should describe the JSON file exported by the VP plugin. However, as discussed in Section 4.2, there are discrepancies between these artefacts and the VP plugin.

For OntoUML to be considered a mature tooling suite, its tools should be compatible, which requires that one OntoUML metamodel acts as a single source of truth. In this way, if a new OntoUML tool is developed, for example, as an alternative to the VP plugin, it should comply with this official metamodel. As a result, it should also work for the transformation developed in this research and other tools using OntoUML JSON files.

We advise to use the OntoUML metamodel developed in this research as a basis for the official OntoUML metamodel because it matches the JSON files currently present in the OntoUML model catalogue. However, the OntoUML developers may consider some of the constructs present in the JSON file as invalid, resulting from bugs in the VP plugin. In this case, the VP plugin as well as the official metamodel need to be adjusted.

Possible transformation adjustment

In case the OntoUML developers decide to change the VP plugin to match a new official metamodel, the transformation developed in this research would no longer be compatible with the VP plugin and thus would require adjustments.

The complexity of adjusting the transformation depends on how the changes deviate from the current JSON files. In all cases, only the Ecore model (described in Section 4.1) and the ATL transformation (described in Section 5.1) need to be adjusted; the Acceleo code generation does not need to be changed as the form of the implementation model would stay the same.

In the best case, only the names of properties in the JSON file would have to be changed. This would only require renaming classes and properties in the OntoUML Ecore metamodel and refactoring the ATL transformation to match these new names. In case more structural changes to the JSON file are made, the necessary refactoring of the ATL transformation may become more complex.

8.2 Improving the validation

The correctness of the transformation, i.e., whether patterns in the source model yield the expected patterns in the target model (as defined by Section 3.3), has mainly been manually tested. This has been done for small custom-made OntoUML fragments as well as a selection of five larger models from the OntoUML model catalogue.

In the automated part of the validation, models from the OntoUML model catalogue have been put through the entire transformation chain and the generated code has been checked for compilation. This still only gives a partial indication of the correctness of the code, as it could be that the generated implementation model was incomplete (i.e., not all elements that should have been generated were generated) but was consistent (i.e., no mismatches of types that result in invalid code).

Ideally, automated test cases should be defined that relate fragments of OntoUML to expected patterns in the generated implementation model, which would give more confidence in the correctness of the implemented transformation. However, this kind of automated testing is currently not supported by ATL. Defining a method to perform automated testing would not only benefit the OntoUML-to-implementation-model transformation but would also be of added value to other ATL transformations. Therefore, we see this as a relevant research direction.

8.3 Generating code for other languages

In this research, we use OntoUML models to generate Java code. However, we also acknowledge that certain users might want to generate code in other object-oriented programming languages. In the design of our transformation chain, we address this by the intermediate step of generating an implementation model. In case one wants to generate code in another language, only the last step which transforms the implementation model to code needs to be changed.

8.3.1 Using Papyrus to generate code from UML

Papyrus[1] is an Eclipse tool for editing UML diagrams. Papyrus Software Designer[2] is a module within Papyrus that allows code to be generated from UML diagrams. Currently, Java, Python, and C++ are supported.

As our implementation model is a UML diagram, in theory, the Papyrus Software Designer could be used to generate code for these languages. However, the Papyrus code

[1] See https://eclipse.dev/papyrus/documentation.html.
[2] See https://gitlab.eclipse.org/eclipse/papyrus/org.eclipse.papyrus-designer/-/wikis/home.

generation functionality is meant to be used for UML models also developed within Papyrus, and not by UML models initialised outside of Papyrus as in our case.

However, as the Papyrus implementation is open-source, this code generation function is available to be reused in some other form. Either the Papyrus tool could be modified to work with models generated from other sources, or the code generation capabilities could be extracted to a standalone UML-to-code generation project.

8.3.2 Defining a new Acceleo UML-to-X transformation

In our research, we used an Acceleo project to generate Java code from an implementation model. If one would like to generate code in another programming language, the seemingly most straightforward option would be to create or reuse another Acceleo project.

In a brief search, two examples of Acceleo projects that generate code from UML diagrams were found. One more extensive project supports code generation for C++[3] and a smaller project was found that generates Python code[4]. However, both projects seem to be developed by individual developers, so no guarantee can be given about the maturity of these projects.

8.3.3 The platform-independence of the implementation model

The notion of an implementation model in this research aimed to be a platform-independent representation for object-oriented languages. However, we only tested code generation for the Java language. As such, the implemented transformation is somewhat tailored to Java, especially in the use of Java naming conventions as described in Section 5.1.1. Although these naming conventions are most often valid in other object-oriented languages (such as Python or C#), not adhering to the naming conventions of a language can be considered a drawback.

C# Besides naming conventions, C# shares a lot of similarities with Java. This is also the case for multiple inheritance features. For example, although C# does not allow multiple class inheritance, it allows the implementation of multiple interfaces[5]. The relevance of this for our transformation is explained in Section 5.4.3.

Python Python differs more from Java when compared to C#. The biggest difference is that Python is a dynamically typed language[6], so that variables do not have explicit types, even though this information is present in our implementation model. A possibility could be to add the type information present in the implementation model as type hints in the generated Python code, as this would make the code more intelligible.

Another interesting difference compared to Java is that Python has no native support for interfaces. This is tricky as our transformation generates UML Interfaces for OntoUML non-sortal types. However, Python does support abstract classes and multiple-class inheritance[7]. One apparent solution to mitigate this is to generate abstract classes instead of interfaces, so that the UML *Interface Realization* should then be interpreted as a class extension in Python. In our implementation model, multiple interface implementations

[3] See https://github.com/vahdat-ab/UML2CPP.
[4] See https://gist.github.com/aranega/f07e4cb4e850af3288af.
[5] See https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/inheritance.
[6] See https://docs.python.org/3/faq/general.html#is-python-a-compiled-language.
[7] See https://docs.python.org/3/library/abc.html for abstract classes and https://docs.python.org/3/tutorial/classes.html#multiple-inheritance for multiple inheritance in Python.

may occur, but as Python allows multiple class inheritance, classes that implement multiple interfaces in the implementation model can simply extend these abstract classes in the Python code. Therefore, we foresee no issues in this regard when generating Python code from the implementation model.

8.4 Implementation model with UML Profile

In the current transformation chain, the implementation model is simply defined as a UML Class diagram along with some restrictions relating to multiple inheritance. However, in earlier stages of this research, we considered the use of a UML Profile to increase the information that could be contained in our implementation model and so increase the semantics transferred to the generated code. Initially, this was done to indicate *'final'* associations, i.e., associations of which one or both ends should be marked as *final* in the generated Java code. In our current transformation, we use the UML *isLeaf* attribute for this (as discussed in Section 3.2).

However, the use of a UML Profile could be useful if one wants to extend the expressiveness of the implementation model. For example, information could be added regarding what forms of constructors need to be generated for a particular class, or, information on how to persist certain objects in a database.

8.4.1 UML Profile in an ATL transformation

In case the implementation model is defined with a UML Profile, the transformation signature of the OntoUML-to-implementation-model transformation would change. Since a UML Profile itself is a UML model, the implementation model profile would become a second input model, as can be seen in the altered transformation signature in Figure 8.1. The resulting target model then is a UML Class diagram to which the UML Profile has been applied.

The ATL support of UML Profiles is minimal. The largest issue is that an ATL transformation is defined on the level of the metamodels. However, the stereotypes defined in a profile are only present at the model instance level (i.e., the *Implementation Model Profile* in Figure 8.1), to which the transformation definition has no direct access (note the arrow with the name *uses* towards the UML2 metamodel). The support of profiles in ATL, along with possible solutions, is further described in [35].

8.5 Persisting data for a generated application

The work presented in this research facilitates the use of OntoUML in software development by generating Java code. Only the code skeleton (i.e., the classes and attributes) is generated, so the required functionality (methods and logic) needs to be added manually. For many use cases, there may be a desire to persist the data in, for example, a relational database.

One option might be to use the generation of relational schemas as described in [13, 14], which generates a database schema from an OntoUML model. However, as discussed in Section 7.1, there is no one-on-one mapping between the generated database schema and the generated Java code, so the relational schema contains fewer tables than generated Java classes. Therefore, a manual solution would be required to link the Java code to this generated schema.

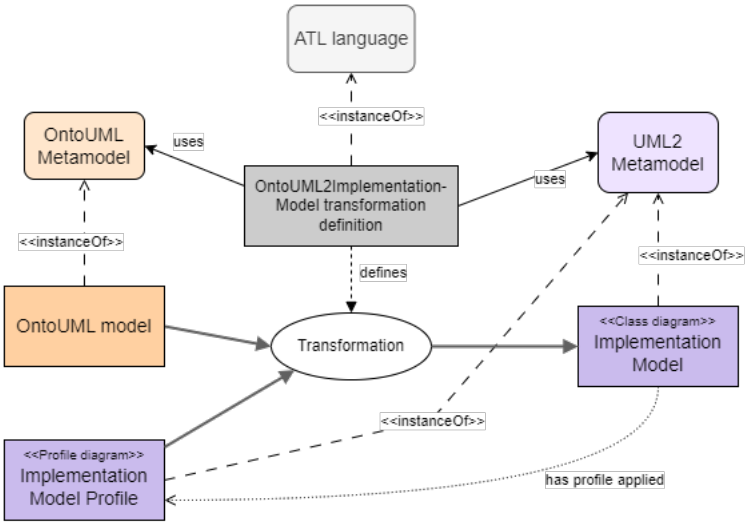


FIGURE 8.1: Transformation signature of ATL transformation in case a UML Profile is applied to the implementation model.

Another option to persist data from the generated code into a database could be to use other tools such as Jakarta persistence[8], in which Java classes can be annotated to create mappings between the Java application and database tables. In case no database schema is defined, Jakarta persistence can generate the relational schema based on the Java classes, thus ensuring correspondence between the generated code and database schema.

[8]See https://jakarta.ee/specifications/persistence/.

Chapter 9

Conclusion

This chapter concludes the work presented in this research. Section 9.1 summarizes the different contributions made. Section 9.2 lists possible directions for future work mainly relating to the evaluation phase of Design Science Research. Finally, Section 9.3 lists recommendations that mainly relate to OntoUML, which refers to different sections in this report that highlight areas of improvement.

9.1 Contributions

The contributions of this research can be summarised as follows. First of all, a frequency analysis of the usage of stereotypes in the OntoUML model catalogue has been performed, which is described in Section 3.1. This reveals the most popular stereotypes, which inspired the selection of OntoUML stereotypes that should be covered by our proposed transformation.

Furthermore, Section 4.1 describes the developed Ecore metamodel for OntoUML that is aligned to the OntoUML VP plugin, along with a method to instantiate this Ecore model from an OntoUML JSON file exported by the VP plugin. This ensures that the developed OntoUML-to-Java transformation can be used for models created with the VP plugin, which is the current most popular tool for developing OntoUML models. Besides this, the integration of OntoUML in EMF enables the use of publicly available tools for other model transformation projects tailored to OntoUML.

As an extension to other work described in [32] and [3] (both described in Chapter 7), we provided a transformation of OntoUML to an implementation model for a wider selection of OntoUML types. For some OntoUML constructs, we made different design decisions for the transformation, either because of considerations in the UFO/OntoUML semantics or the pragmatic effects when generating an implementation model tailored to Java.

Our proposed OntoUML-to-implementation-model transformation has been implemented in ATL. The generation of an implementation model that is defined in UML, which is a publicly available and widely used modelling language, enables the possible generation of code in other programming languages, which is further discussed in Section 8.3.

In our transformation chain, we decided to generate Java code, which is achieved by an Acceleo project that takes a UML model as input. As described in Section 5.4, we reused an existing but archived project created by the Acceleo developers. We adjusted this project to work with Acceleo version 3.7.17[1] and fixed some found bugs. This revamped Acceleo UML to Java project can also be used for other EMF projects that work with UML.

[1] During our research, Acceleo version 4 has been released which is not directly compatible with version 3.

Finally, both manual and automated validation have been performed on a selection of models from the OntoUML model catalogue, described in Chapter 6. The automated validation indicates that the OntoUML-to-Java transformation yields valid Java code for 70% of the transformed models. For the cases in which the generated code failed this test, the cause was analysed. In most cases, the issue in the generated code was caused by what we consider invalid constructs in the source model. In an additional validation effort, we manually analysed a selection of five OntoUML models for which the generated Java code for each OntoUML model element was inspected, giving a reasonable indication that the implemented transformation works as intended. Both the automated and manual validation also exposed constructs present in the public models that would be deemed invalid by the OntoUML VP plugin.

9.2 Future work

In this research, a transformation of OntoUML to Java has been implemented and validated on a technical level, i.e., the step 'demonstration' in Design Science Research (illustrated in Figure 1.1). However, the evaluation step has not been performed, which would require employing the transformation in the context of the development of an application, yielding insights on how to improve the transformation.

Another qualitative evaluation could entail a comparison of the generated code for an OntoUML model with code written by a developer for the same OntoUML model. This could give insights into the validity of the proposed transformation rules as well as indicate whether the generated code can be easily understood by a developer responsible for further implementing the transformation.

Furthermore, the claim has been made that the use of OntoUML yields higher-quality models compared to plain UML class diagrams [33], and therefore the expectation is that software applications based on such ontologies are of higher quality as well [28]. This expectation might be confirmed by applying our developed transformation in practice. Ideally, an experiment could be performed that describes a case for a to-be-developed application, which is to be implemented by two developers/groups of developers. One group defines a UML model and uses a traditional code generator for UML[2], whereas the other group defines an OntoUML model and uses the proposed OntoUML-to-Java transformation. Both developed applications can be evaluated according to software quality metrics and/or competency questions made for the case. Besides these evaluations, both developer groups can be interviewed about their experiences about the ease of development.

9.3 Recommendations

This research yielded several insights mainly related to the implementation of OntoUML.

9.3.1 Alignment of OntoUML tools

Inconsistencies were observed between the OntoUML VP plugin, the platform-independent metamodel and the JSON Schema (further discussed in Section 8.1). We would advise to alter the platform-independent metamodel and JSON Schema to match the developed Ecore metamodel as described in Section 4.1, which was made based on the JSON exported by the VP plugin. Since all the JSON files present in the model catalogue have

[2]For example either the implementation-model-to-Java Acceleo transformation described in Section 5.4 or the Papyrus tool described in Section 8.3.

apparently been created with the VP plugin; they match the Ecore metamodel we developed. Therefore, it is likely that other tools that use OntoUML have also been matched to these JSON files.

The changes required in the platform-independent metamodel are relatively small and are described in Section 4.2.2. However, we observed larger structural changes with the JSON Schema, which is why we simply marked it as 'incorrect' and disregarded it. However, this would require a more detailed inspection. We still deem it valuable to have a working JSON Schema, as this allows for JSON files that originate from different tools to be validated. In this research, we took the JSON output by the VP plugin as the single source of truth, however, if other tools also generate JSON files (with possibly a different structure), it would be useful to have one artefact (the JSON Schema) that defines the structure of correct OntoUML JSON files.

9.3.2 Extended model validation

OntoUML provides a model validation tool that is integrated into the VP plugin, which allows checking the correctness of a model. We found that this validation does not cover all constructs that we deem necessary for our transformation to work correctly. Section 5.3 describes these three extra assumptions we impose on the model: no properties without types, no duplicate generalizations, and no multiple memberOf relations.

The 'no properties without types' assumption is mainly important in the context of generating code: if a variable is defined, the type should be known[3]. However, when an OntoUML model is used as a communication vehicle between people, it may not always be necessary to define the type of a variable, as knowing such a property exists without knowing the exact type might be sufficient.

However, for the 'no duplicate generalizations' and 'no multiple memberOf relations' assumptions, we think a model violating them is incorrect. The violation of these assumptions is not checked by the VP plugin model validator. Because of a lack of a single source of OntoUML specification, we are not certain whether these constructs are invalid in OntoUML. Furthermore, we acknowledge that it may not be desirable to add these constraints for all OntoUML models, since, as we have stated, OntoUML models can be used for other purposes than generating code. However, if these constraints are recognised by the OntoUML/UFO developers as invalid OntoUML constructs, it would be an added value to also check for these constraints in the VP plugin to help modellers create better models.

9.3.3 Missing property values in the OntoUML JSON file

We found two properties that are missing in the JSON file exported by the VP plugin that are relevant for our implemented transformation:

- The *'stereotype'* property in the OntoUML Property element, whose values can be *'begin'* or *'end'*, seem to correspond with properties of relations and the relation direction. However, these are not set as such by the VP plugin. For our transformation, we need to to the direction of a relation, which is described in more detail in Section 5.1.3. Although some ways have been found to work around this, we would advise to explicitly set the direction of the transformation in the JSON file.

[3]This is at least relevant for languages requiring the type of a variable to be known at the declaration. For a language such as Python, this is not required.

- The *'redefinedProperties'* field of the Property element. As described in Section 6.2.3, one model was found that makes use of this construct to redefine elements, however, this functionality seems to not be supported by the VP plugin as the value to this property is always set to null. If such constructs are to be supported by an OntoUML-to-Java transformation, these should be set in the OntoUML JSON file.

9.3.4 Selection criteria OntoUML model catalogue

We have found several models in the OntoUML model catalogue that do not pass the validation of the OntoUML VP plugin but are not marked as invalid. When developing a tool that is tested with models from the catalogue, an error might be due to a faulty implementation of the tool (e.g., the transformation developed in this research) but might also be the result of an invalid OntoUML model. The actual cause of the fault had to be manually checked.

Alternatively, one could consider only admitting models to the catalogue that pass the VP plugin validation. However, a motivation of the catalogue maintainers could be to include as many examples of OntoUML models despite some having (minor) issues. In this case, an improvement could be to add a flag to the model metadata in the catalogue on whether this model passes the VP plugin validation, which can be used by users to filter models if they need to.

Appendix A

Statistics on OntoUML stereotypes

Table A.1: The occurrence count of class stereotypes in the OntoUML Model Catalog.

Stereotype	Occurs in # projects	Total occurrences	Valid stereotype
kind	136	1843	True
relator	125	1405	True
role	116	2180	True
subkind	116	2215	True
category	96	611	True
roleMixin	65	571	True
mode	63	552	True
collective	58	184	True
phase	56	383	True
event	49	447	True
quality	42	215	True
null	34	786	False
datatype	34	118	True
type	26	212	True
mixin	23	85	True
situation	19	78	True
quantity	13	45	True
enumeration	9	69	True
phaseMixin	8	20	True
object	3	5	False
historicalRoleMixin	3	9	True
historicalRole	3	22	True
disposition	3	7	False

Continued on next page

Table A.1: The occurrence count of class stereotypes in the OntoUML Model Catalog. (Continued)

Stereotype	Occurs in # projects	Total occurrences	Valid stereotype
nonPerceivableQuality	2	2	False
participation	2	18	False
proposition	2	5	False
abstract	2	2	True
stringNominalStructure	2	2	False
Goal	1	951	False
Document	1	48	False
Resource	1	9	False
Plan	1	15	False
Softgoal	1	18	False
Actor boundary	1	7	False
Normative Description	1	6	False
Actor	1	64	False
Normative Document	1	85	False
CQ	1	217	False
atomic event	1	4	False
NonPerceivableQuality	1	5	False
processual role	1	7	False
material relation	1	3	False
Proposition	1	3	False
Agent	1	2	False
commitment	1	2	False
trope	1	4	False
Complex Event	1	2	False
set	1	2	False
TimePoint	1	2	False
UFO-C	1	2	False
abstract individual	1	1	False
Activity	1	1	False
agent	1	1	False
belief	1	1	False
ComplexAction	1	1	False
ComplexEvent	1	1	False

Continued on next page

Table A.1: The occurrence count of class stereotypes in the OntoUML Model Catalog. (Continued)

Stereotype	Occurs in # projects	Total occurrences	Valid stereotype
endurant	1	1	False
goal	1	1	False
HumanAgent	1	1	False
InstitutionalAgent	1	1	False
intention	1	1	False
MentalMode	1	1	False
Organization	1	1	False
Processual Role	1	1	False
quale	1	1	False
quality dimension	1	1	False
quality structure	1	1	False
service	1	1	False
UFO-B	1	1	False
viewpoint	1	1	False

Appendix B

Automated validation on OntoUML model catalogue models

B.1 Specification of execution environment

TABLE B.1: Specifications of the execution environment used to execute the OntoUML to Java transformation.

Operating System	Windows 11 x64
Processor	13th Gen Intel Core i7-12700H, 2400 Mhz, 14 Cores, 20 Logical Processors
Memory (RAM)	32 GB
Java JDK	graalvm-jdk-21.0.2

B.2 Results

This section lists the results of the performed automated validation on models from the OntoUML model catalogue. Table B.2 shows the results for the 79 models that successfully went through the entire transformation and thus yielded Java code. The column *# Classes* contains the size of each model in number of classes. The column *Time* contains the measured execution time of the transformation in seconds. The column *ATL warnings* contains the warnings that were given during the ATL transformation, these are further described in Section 6.2.3. The column *Compiles* indicates whether the generated Java code could be compiled.

Table B.3 shows the different kinds of compile errors that occurred for the 24 models that generated faulty Java code. The column *Compile error category* contains the type of compile error that occurred, the four possible categories are explained in Section 6.2.3. The column *Error detail* gives information where in the generated code the error occurs. For the categories *Multiple class inheritance present* and *Reserved keyword name*, the name of the class in which the error occurs is included. For the categories *Duplicate variable definition* and *Cannot override from super*, the name of the attribute/variable as well as the class is included.

Table B.2: Transformation results for the 79 models that successfully pass the transformation.

Model	# Classes	Time (s)	ATL warnings	Compiles
aguiar2018rdbs-o	43	16.05		True
aguiar2019ooco	56	16.26		False
bank-account2013	18	15.94	Contains property without type	True
bank-model	23	16.06	Contains property without type	True
barcelos2013normative-acts	45	16.22	Contains property without type	True
barcelos2015transport-networks	487	20.99	Contains property without type	True
barros2020programming	12	15.81		True
bernasconi2023fair-principles	46	16.32		True
brazilian-governmental-organizational-structures	15	15.91		True
buchtela2020connection	19	16.05		True
buridan-ontology2021	54	16.44	Contains property without type	False
carolla2014campus-management	17	16.07		True
castro2012cloudvul-nerability	32	16.2		True
cgts2021sebim	29	16.07	Contains property without type	True
clergy-ontology	29	16.24		True
construction-model	13	15.84		True
demori2023miscon	31	16.09		False
digitaldoctor2022	48	16.51	Contains property without type	False
elikan2018brand-identity	30	16.22		False
eu-rent-refactored2022	66	16.53	Contains property without type	True
experiment2013	22	16.05		True
formula-one2023	26	16.11		True
fraller2019abc	39	16.26	Contains property without type, Contains empty string	False

Continued on next page

Table B.2: Transformation results for the 79 models that successfully pass the transformation. (Continued)

Model	# Classes	Time (s)	ATL warnings	Compiles
fumagalli2022criminal-investigation	10	16.01		True
g809-2015	24	16.1		True
genealogy2013	8	15.89		True
gomes2022digital-technology	19	16.03		False
guarino2016value	18	16.18		True
haridy2021egyptian-e-government	32	16.18		True
health-organizations	24	16.0		True
idaf2013	46	17.27		False
internal-affairs2013	61	16.93		True
internship	26	16.47	Contains property without type	True
it-infrastructure	31	16.58		False
jacobs2022sdpontology	34	16.59	Contains property without type	False
junior2018o4c	11	16.22	Contains property without type	False
khantong2020ontology	30	16.96		True
laurier2018rea	23	16.57		True
library	43	16.84		False
machacova2023gym	41	16.67	Contains property without type	True
martinez2013human-genome	10	16.27		True
music-ontology	42	16.97	Contains GeneralizationSet, Contains property without type	False
neves2021grain-production	18	16.3		True
online-mentoring	29	16.48		True
pereira2015doacao-orgaos	25	16.48		True
pereira2020ontotrans	35	16.57		False
photography	18	16.27		False
plato-ontology2019	73	17.08	Contains property without type	False

Continued on next page

Table B.2: Transformation results for the 79 models that successfully pass the transformation. (Continued)

Model	# Classes	Time (s)	ATL warnings	Compiles
project-assets2013	19	16.55	Contains property without type	True
project-management-ontology	14	16.38		True
public-expense-ontology2020	42	16.77		True
public-organization2013	46	16.91		False
public-tender	71	17.34	Contains property without type	False
qam	41	16.95		True
ramirez2015userfeedback	50	17.04		True
real-estate-ontology	15	16.48		True
recommendation-ontology	18	16.49	Contains property without type	True
road-accident2013	15	16.32	Contains property without type	True
rocha2023ciencia-aberta	24	16.41		True
santos2020valuenetworks	15	16.69		True
scientific-experiment2013	20	16.5		True
scientific-publication2013	55	16.83	Contains property without type	True
short-examples2013	35	16.78		True
silva2012itarchitecture	84	17.35	Contains property without type	True
silva2021sebim	42	16.77	Contains property without type	False
silveira2021oap	32	16.77		True
social-contract	33	16.59		True
social-contracts2013	33	16.6		True
sousa2022triponto	44	16.87		True
srro-ontology	20	16.47	Contains property without type	False
stock-broker2021	14	16.57		True

Continued on next page

Table B.2: Transformation results for the 79 models that successfully pass the transformation. (Continued)

Model	# Classes	Time (s)	ATL warnings	Compiles
telecom-equipment2013	34	16.79		True
tender2013	84	17.1	Contains property without type	True
tourbo2021	49	16.84	Contains property without type	False
university-ontology	32	16.65		False
valaski2020medical-appointment	7	16.53		True
zanetti2019orm-o	34	16.79		False
zhou2017hazard-ontology-robotic-strolling	13	16.78		True
zhou2017hazard-ontology-train-control	25	16.69		False

Table B.3: Compile error category per model.

Model	Compile error category	Error detail
aguiar2019ooco	Multiple class inheritance present	AbstractClass
buridan-ontology2021	Multiple class inheritance present	MaterialSupposition
demori2023miscon	Reserved keyword name	CommDevice
digitaldoctor2022	Multiple class inheritance present	DiagnosticProcedure
elikan2018brand-identity	Duplicate variable definition	variable members in Stakeholders
fraller2019abc	Duplicate variable definition	variable activityLevel in Activity
gomes2022digital-technology	Duplicate variable definition	variable technicalIdentityOfObjects in TechnicalIdentityOfObject
idaf2013	Duplicate variable definition	variable members in TechnicalAdministrativeSupport
it-infrastructure	Duplicate variable definition	variable members in ITTeam
jacobs2022sdpontology	Duplicate variable definition	variable role in BusinessRole
junior2018o4c	Duplicate variable definition	variable relationship in Concern
library	Duplicate variable definition	variable members in Collection

Continued on next page

Table B.3: Compile error category per model. (Continued)

Model	Compile error category	Error detail
music-ontology	Cannot override from super	attribute getMembers in classes MusicalGroup and ArtisticGroup
pereira2020ontotrans	Duplicate variable definition	variable getHelp in Informativeness
photography	Duplicate variable definition	variable members in Archive
plato-ontology2019	Duplicate variable definition	variable members in ProducerPerson
public-organization2013	Duplicate variable definition	variable members in ApoioTecnicoAdministrativo
public-tender	Duplicate variable definition	variable getPublication in TenderInCallForTender
silva2021sebim	Duplicate variable definition	variable lightAnalysis in Construction
srro-ontology	Reserved keyword name	ClassTestCase
tourbo2021	Duplicate variable definition	variable create in Plan
university-ontology	Reserved keyword name	ClassEnrollment
zanetti2019orm-o	Reserved keyword name	EntityClass
zhou2017hazard-ontology-train-control	Duplicate variable definition	variable provider in Communication

B.3 Cause of 'duplicate variable definition' compilation error

Section 6.2.3 describes the occurrence of a compilation error that might occur in code that is generated with the OntoUML-to-implementation-model transformation. This section describes the causes of this issue for seven models which did were not described in Section 6.2.3.

B.3.1 Duplicate relation end names

Four of the models contained classes that have two or more relations that have the same outgoing relation end name. This results in classes being generated for these OntoUML types that contain multiple references with the same name.

gomes2022digital-technology Figure B.1 displays a section of the *gomes2022digital-technology* model. In the generated code of the *Technical Identity of Object* relator, a duplicate variable is defined with the name *technicalIdentityOfObjects*. At first glance, the included image of the model (the upper part of Figure B.1) does not make clear where this variable originates from. However, when having a closer look at the relation properties of the *physical form* and *social funtion* relations, both have a relation end name *'technical Identity of Objects'* which is the cause of this duplicate variable appearing in the generated code (as can be seen in the lower part of Figure B.1).

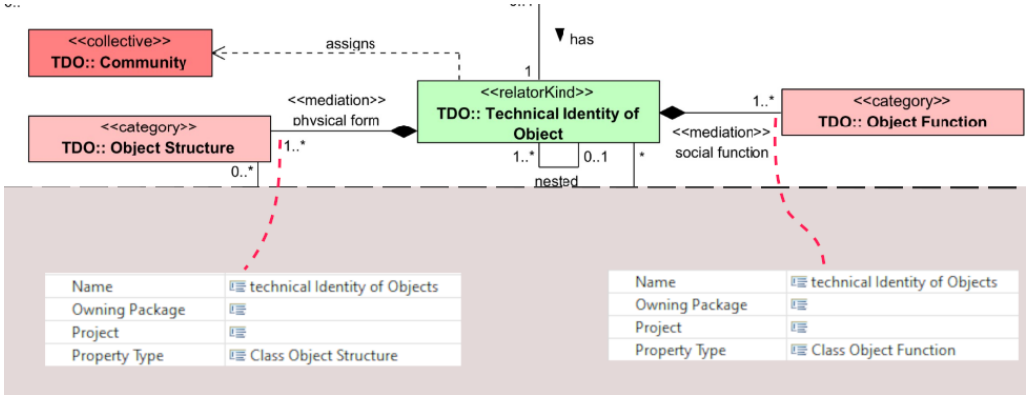


FIGURE B.1: Fragment of the gomes2022digital-technology model along with the properties of two relation ends, indicating duplicate property names.

It should be noted that there seems to be a mismatch between the included image of this model in the OntoUML model catalogue and the included JSON file, as these relation end names are not visible in the image. Furthermore, the *Technical Identity of Object* has the invalid stereotype *relatorKind* in the image while the correct stereotype *relator* is used in the JSON file.

tourbo2021 Figure B.2 displays part of the *tourbo2021* model. In this case, the relator *plan* has two relations that both have a relation end named *'create'*. In the image of the diagram included in the model catalogue, the name of this relation end seems to be moved towards the place where normally the relation name would be. However, the JSON indicates that these names actually belong to the relation end properties, as indicated by the red dashed lines added in Figure B.2.

Pereira2020ontotrans Figure B.3 displays part of the *pereira2020ontotrans* model, which shows a roleMixin *Informativeness* that is connected to many other classes with relations. All relations seem to be called *'help'*, however, when looking into the included JSON file, two of these names are attached to the relation end properties, namely those for the classes *Correctness* and *Consistency* (these are highlighted in Figure B.3).

public-tender Figure B.4 displays part of the *public-tender* model. The role *Tender in Call for Tender* is related to the relator *Call for Tender publication* of which the relation end has the name *'publication'*. Furthermore, the roleMixin *Publication Context* is related to the relator *Publication* also with a relation end named *'publication'*.

As *Tender in Call for Tender* specializes the roleMixin *Publication Context*, in the resulting implementation model, the reference *publication* from *Publication Context* is inherited by *Tender in Call for Tender*, which in turn results in a duplicate method definition in the generated code.

B.3.2 Duplicate relations not visible in the diagram images

In two models, constructs occurred in which multiple relations are present between two classes corresponding to the limitation of the transformation described in Section 5.2.2. In contrast to the models *junior2018o4c* and *fraller2019abc* (as discussed in Section 6.2.3), the models described in this section seem to unintentionally include this feature.

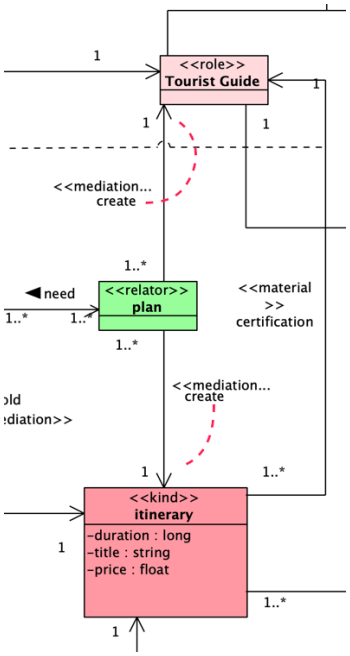


FIGURE B.2: Fragment of the tourbo2021 OntoUML model that displays two relations with duplicate relation end names.

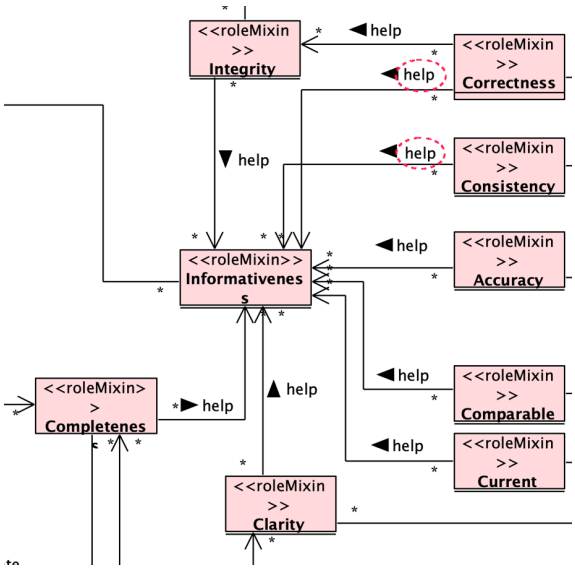


FIGURE B.3: Fragment of the pereira2020ontotrans model.

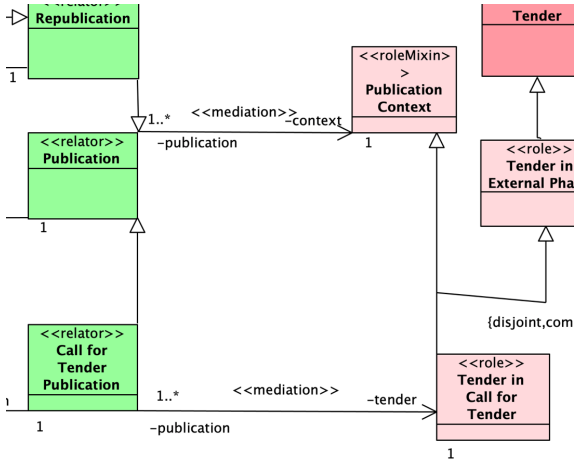


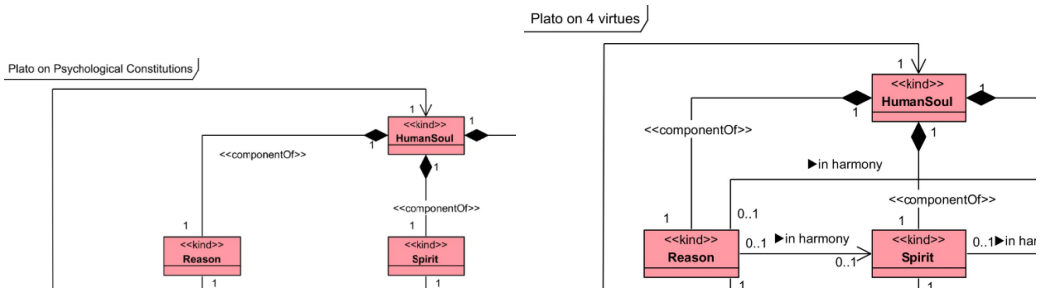
FIGURE B.4: Fragment of the public-tender OntoUML model.

plato-ontology2019 The *plato-ontology2019* contains multiple images displaying various packages of different viewpoints regarding the works of Plato. Two fragments of this model belonging to seemingly different packages can be seen in Figure B.5. Both fragments include the kinds *HumanSoul* and *Reason*, both with a *componentOf* relation connecting them. The names listed in the top-left corner of these images (*'Plato on Psychological Constitutions'* and *'Plato on 4 virtues'*) seem to indicate that these kinds are part of different packages. E.g., a distinct occurrence of *HumanSoul* exists in the package *'Plato on Psychological Constitutions'* as well as in *'Plato on 4 virtues'*. However, the JSON file included for this model does not separate these classes into different packages.

The result is that the JSON contains only one occurrence of both *HumanSoul* and *Reason* but still contains two separate *componentOf* relations connecting these classes.

silva2021sebim The model *silva2021sebim* seems to contain a similar issue. Figure B.6 displays a part of this model, in which the relation between the classes *LightAnalysis* and *Construction* is highlighted. In this image of the diagram included in the model catalogue, only one relation between these classes is visible. However, the JSON file contains two instances of this relation.

A possible explanation for this is that the creator of the model at one point deleted



(A) Caption (B)

FIGURE B.5: Two fragments showing of the *plato-ontology2019* model showing multiple occurrences of the kinds *HumanSoul* and *Reason* which seemingly are located in different packages.

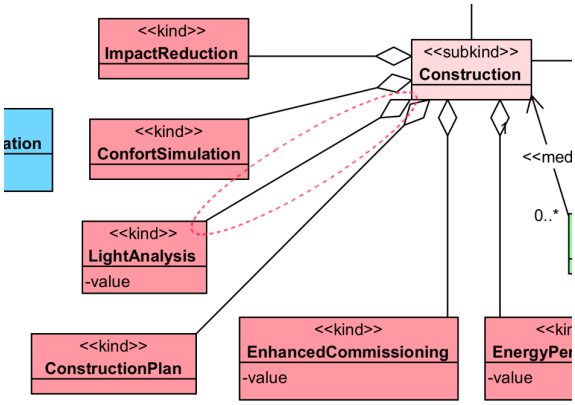


FIGURE B.6: Fragment of the *silva2021sebim* model in which an relation that is included twice in the JSON file is highlighted.

this relation from the diagram, but not from the underlying model repository[1]. At a later point, the creator might have decided the relation should be present and thus added a new relation with as result that the underlying model contains two instances of that relation.

B.3.3 Special case of the train-control ontology

The *zhou2017hazard-ontology-train-control* model, of which an image is displayed in Figure B.7, is a bit of a special case. Seemingly, multiple classes with the same name appear, such as the relator *Communication* and the role *Provider*. In the included JSON file, there is only one instance of each of these classes, which is expected as Visual Paradigm requires different classes to have unique names. However, the image also displays two relations between *Communication* and *Provider* which occur at two separate places. These relations are included as two distinct relations in the JSON file, thus resulting in a duplicate relation between these two classes.

Besides this, the model is also considered invalid as the roles do not specialize sortals, and thus this model would also not pass the validation check by the OntoUML VP plugin.

[1]When deleting a model element in Visual Paradigm which does not appear in another diagram, the user is asked whether to delete the model element only from the diagram or also from the underlying repository.

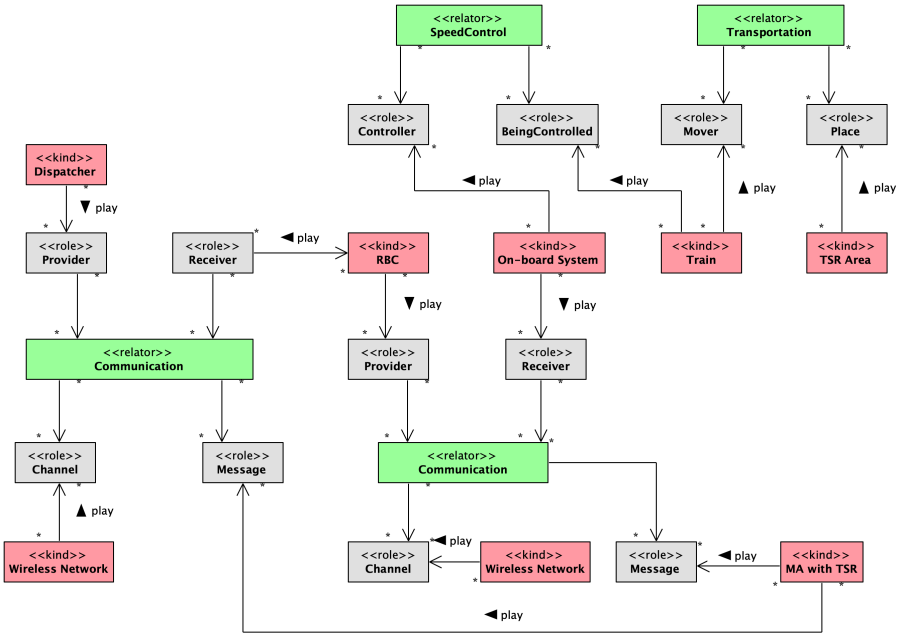


FIGURE B.7: Fragment of the *zhou2017hazard-ontology-train-control* model from the OntoUML model catalogue

Appendix C

Manual validation checklist

Figure C.1 and Figure C.2 include the checklist that has been used for the manual validation of the transformation. It consists of questions that should be asked about the generated Java code, grouped for elements that can be found in the OntoUML model. Note that one question may apply to multiple occurrences of a specific construct in the OntoUML model.

The workflow thus is to go over every element in the OntoUML model (either class, relation, or generalization), and find the applicable categories in the checklist. For instance, for an OntoUML model containing a class marked with the stereotype *kind*, the questions under the headers *Classes in general* and *For identity providers* should be answered. The question is marked affirmative (i.e., 'Yes') if it can be answered as such for every applicable model element in the OntoUML model. The question is marked as failing (i.e., 'No') if the answer to the question is no for at least one element in the OntoUML model. The box *does not apply* is checked in case the question is not applicable to the analysed model, which could be the case if, for example, the model does not contain a *memberOf* relation.

C.1 Filled-in checklists

This section includes all the filled-in checklists for each of the validated OntoUML models. Along with the filled-in checklist with nodes, a screenshot of the ontology with annotations of the checked elements is included. The Java code that was generated is included on https://github.com/GuusVink/ontouml-java-generation.

Validation checklist

OntoUML model: _____

		Passes	Fails	Does not apply
Classes in general				
	Are properties with types taken over into the generated class?			
Relations in general				
	For supported relation stereotypes, is a reference attribute present to the other class in both classes that are connected by the relation?			
	If the upperbound of the cardinality is > 1, is the reference attribute wrapped in the appropiate collection type?			
	Is the name of the reference attribute set to the type of the end (in case no relation end name is present), or to the relation end name in case one is present?			
For identity providers				
	Is a class generated with the appropiate name?			
For generalizations (of roles)				
	Is a reference attribute pointing to the role present in the generic class, and vice versa?			
	Is the reference attribute pointing towards the generic class marked as final?			
	Are both reference attributes not wrapped in a collection type?			
For generalizations (of phases)				
	Check if complex or simple scenario should apply			
	For simple scenario:			
	Is an enumeration generated which contains all phases as literals?			
	Is a reference attribute to this enumeration included in the generic class?			
	For complex scenario:			
	Is an interface generated for the phase partition?			
	Is a reference attribute generated in both the generic class and interface class?			
	Is the reference attribute in the interface marked as final?			
For generalizations (not connecting roles/phases to identitty providers)				
	If between to sortals: does the specific class extend the generic class?			

FIGURE C.1: First part of the validation checklist.

		Passes	Fails	Does not apply
	If between to non-sortals: does the specific interface extend the generic interface?			
	If between sortal and non-sortal: does the specific class implement the generic interface?			
For memberOf relations				
	Is the name of the reference attribute pointing to the member set to 'members' in case no relation end name is present? Otherwise, is that relation name used?			
For mediation relations				
	Is the reference attribute in the class corresponding to the relator marked as final?			
	Is no reference attribute generated for a (possibly present) material relation?			
For characterization relations				
	In case the moment end is a quality: are both involved reference attributes marked as final?			
	In case the moment end is a mode: is only the reference attribute pointing towards the bearer marked as final?			

FIGURE C.2: Second part of the validation checklist.

Validation checklist

OntoUML model: aguiar 2018 rdbs-o

		Passes	Fails	Does not apply
Classes in general				
	Are properties with types taken over into the generated class?			X
Relations in general				
	For supported relation stereotypes, is a reference attribute present to the other class in both classes that are connected by the relation?	X		
	If the upperbound of the cardinality is > 1, is the reference attribute wrapped in the appropiate collection type?	X		
	Is the name of the reference attribute set to the type of the end (in case no relation end name is present), or to the relation end name in case one is present?			X
For identity providers				
	Is a class generated with the appropiate name?	X		
For generalizations (of roles)				
	Is a reference attribute pointing to the role present in the generic class, and vice versa?	X		
	Is the reference attribute pointing towards the generic class marked as final?	X		
	Are both reference attributes not wrapped in a collection type?	X		
For generalizations (of phases)				
	Check if complex or simple scenario should apply			X
	For simple scenario:			
	Is an enumeration generated which contains all phases as literals?			X
	Is a reference attribute to this enumeration included in the generic class?			X
	For complex scenario:			
	Is an interface generated for the phase partition?			X
	Is a reference attribute generated in both the generic class and interface class?			X
	Is the reference attribute in the interface marked as final?			X
For generalizations (not connecting roles/phases to identitty providers)				
	If between to sortals: does the specific class extend the generic class?	X		

FIGURE C.3: First page of the filled-in checklist for the model aguiar2018rdbs-o

		Passes	Fails	Does not apply
	If between to non-sortals: does the specific interface extend the generic interface?	X		
	If between sortal and non-sortal: does the specific class implement the generic interface?	X		
For memberOf relations				
	Is the name of the reference attribute pointing to the member set to 'members' in case no relation end name is present? Otherwise, is that relation name used?			X
For mediation relations				
	Is the reference attribute in the class corresponding to the relator marked as final?			X
	Is no reference attribute generated for a (possibly present) material relation?			X
For characterization relations				
	In case the moment end is a quality: are both involved reference attributes marked as final?			X
	In case the moment end is a mode: is only the reference attribute pointing towards the bearer marked as final?			X

* Contains invalid OntoUML
  ↳ subkind not specializing identity provider
  ↳ mode specializing mode

* Contains non-sortals with associations
  ↳ properly generates get/set prototypes

FIGURE C.4: Second page of the filled-in checklist for the model aguiar2018rdbs-o

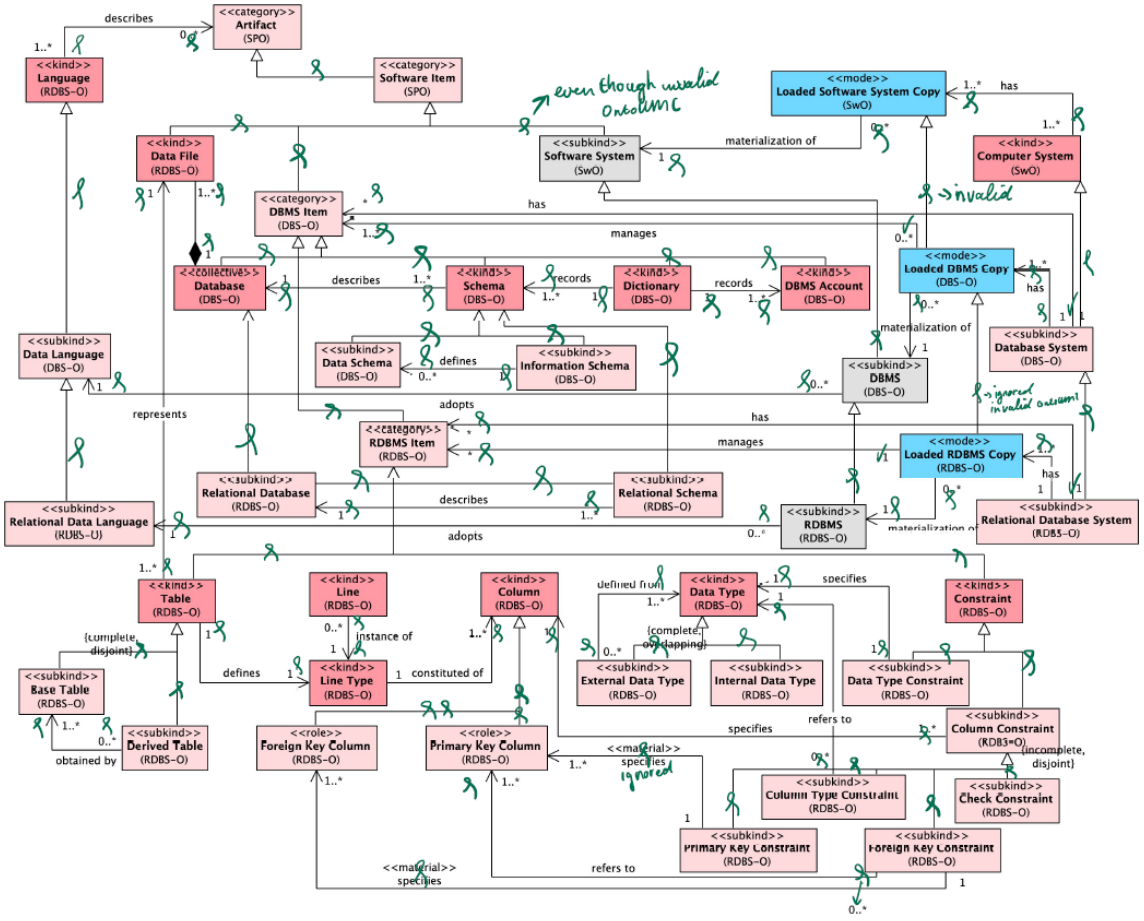


FIGURE C.5: Diagram of the aguiar2018rdbs-o model annotated for manual validation

Validation checklist

OntoUML model: bank_account 2013

		Passes	Fails	Does not apply
Classes in general				
	Are properties with types taken over into the generated class?			X
Relations in general				
	For supported relation stereotypes, is a reference attribute present to the other class in both classes that are connected by the relation?	√		
	If the upperbound of the cardinality is > 1, is the reference attribute wrapped in the appropiate collection type?	√		
	Is the name of the reference attribute set to the type of the end (in case no relation end name is present), or to the relation end name in case one is present?	√		
For identity providers				
	Is a class generated with the appropiate name?	√		
For generalizations (of roles)				
	Is a reference attribute pointing to the role present in the generic class, and vice versa?	√		
	Is the reference attribute pointing towards the generic class marked as final?	√		
	Are both reference attributes not wrapped in a collection type?	√		
For generalizations (of phases)				
	Check if complex or simple scenario should apply			X
	For simple scenario:			
	Is an enumeration generated which contains all phases as literals?			X
	Is a reference attribute to this enumeration included in the generic class?			X
	For complex scenario:			
	Is an interface generated for the phase partition?			X
	Is a reference attribute generated in both the generic class and interface class?			X
	Is the reference attribute in the interface marked as final?			X
For generalizations (not connecting roles/phases to identitty providers)				
	If between to sortals: does the specific class extend the generic class?	√		

FIGURE C.6: First page of the filled-in checklist for the model bank-account2013.

		Passes	Fails	Does not apply
	If between to non-sortals: does the specific interface extend the generic interface?	√		
	If between sortal and non-sortal: does the specific class implement the generic interface?	√		
For memberOf relations				
	Is the name of the reference attribute pointing to the member set to 'members' in case no relation end name is present? Otherwise, is that relation name used?			X
For mediation relations				
	Is the reference attribute in the class corresponding to the relator marked as final?	√		
	Is no reference attribute generated for a (possibly present) material relation?	√		
For characterization relations				
	In case the moment end is a quality: are both involved reference attributes marked as final?			X
	In case the moment end is a mode: is only the reference attribute pointing towards the bearer marked as final?			X

* properties of type 'int' do not appear in the JSON only in image
  (property appears but type is null)
  ↳ seems to be an issue with (an older version of) the vp plugin → could not be replicated

* Contains invalid OntoUML
  ↳ Relator specializing relator

* General issue: does not do well with the Portugese language
  ↳ Special characters are removed

* Found and fixed bug: Roles may extend Roles

FIGURE C.7: Second page of the filled-in checklist for the model bank-account2013.

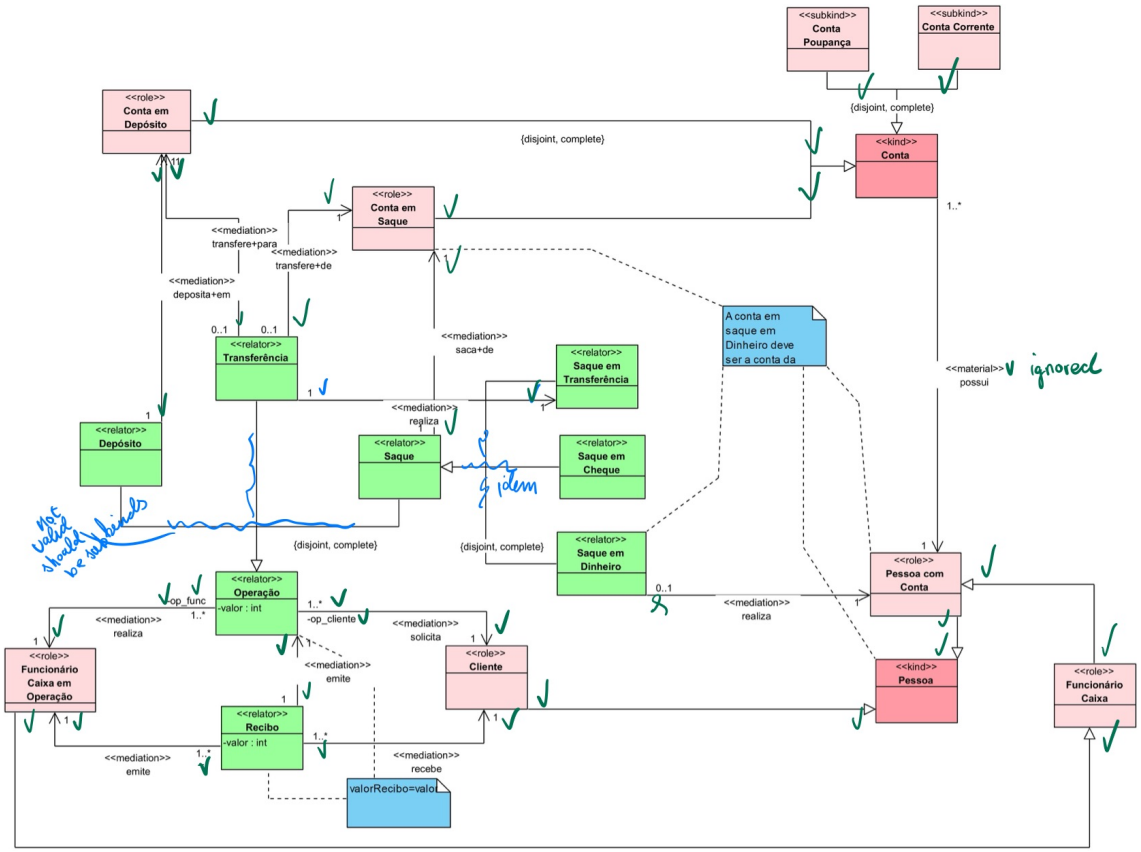


FIGURE C.8: Diagram of the bank-account2013 model annotated for manual validation.

Validation checklist

OntoUML model: bank-model

		Passes	Fails	Does not apply
Classes in general				
	Are properties with types taken over into the generated class?			X
Relations in general				
	For supported relation stereotypes, is a reference attribute present to the other class in both classes that are connected by the relation?	✓		
	If the upperbound of the cardinality is > 1, is the reference attribute wrapped in the appropiate collection type?	✓		
	Is the name of the reference attribute set to the type of the end (in case no relation end name is present), or to the relation end name in case one is present?	✓		
For identity providers				
	Is a class generated with the appropiate name?	✓		
For generalizations (of roles)				
	Is a reference attribute pointing to the role present in the generic class, and vice versa?	✓		
	Is the reference attribute pointing towards the generic class marked as final?	✓		
	Are both reference attributes not wrapped in a collection type?	✓		
For generalizations (of phases)				
	Check if complex or simple scenario should apply			X
	For simple scenario:			
	Is an enumeration generated which contains all phases as literals?			X
	Is a reference attribute to this enumeration included in the generic class?			X
	For complex scenario:			
	Is an interface generated for the phase partition?			X
	Is a reference attribute generated in both the generic class and interface class?			X
	Is the reference attribute in the interface marked as final?			X
For generalizations (not connecting roles/phases to identitty providers)				
	If between to sortals: does the specific class extend the generic class?	✓		

FIGURE C.9: First page of the filled-in checklist for the model bank-model.

		Passes	Fails	Does not apply
	If between to non-sortals: does the specific interface extend the generic interface?	√		
	If between sortal and non-sortal: does the specific class implement the generic interface?	√		
For memberOf relations				
	Is the name of the reference attribute pointing to the member set to 'members' in case no relation end name is present? Otherwise, is that relation name used?	√		
For mediation relations				
	Is the reference attribute in the class corresponding to the relator marked as final?	√		
	Is no reference attribute generated for a (possibly present) material relation?	√		
For characterization relations				
	In case the moment end is a quality: are both involved reference attributes marked as final?			X
	In case the moment end is a mode: is only the reference attribute pointing towards the bearer marked as final?	√		

* Contains invalid OutoUML
  ↳ Relator specialising relator

* Contains non-sortals with associations
  ↳ get/set prototypes generated

FIGURE C.10: Second page of the filled-in checklist for the model bank-model.

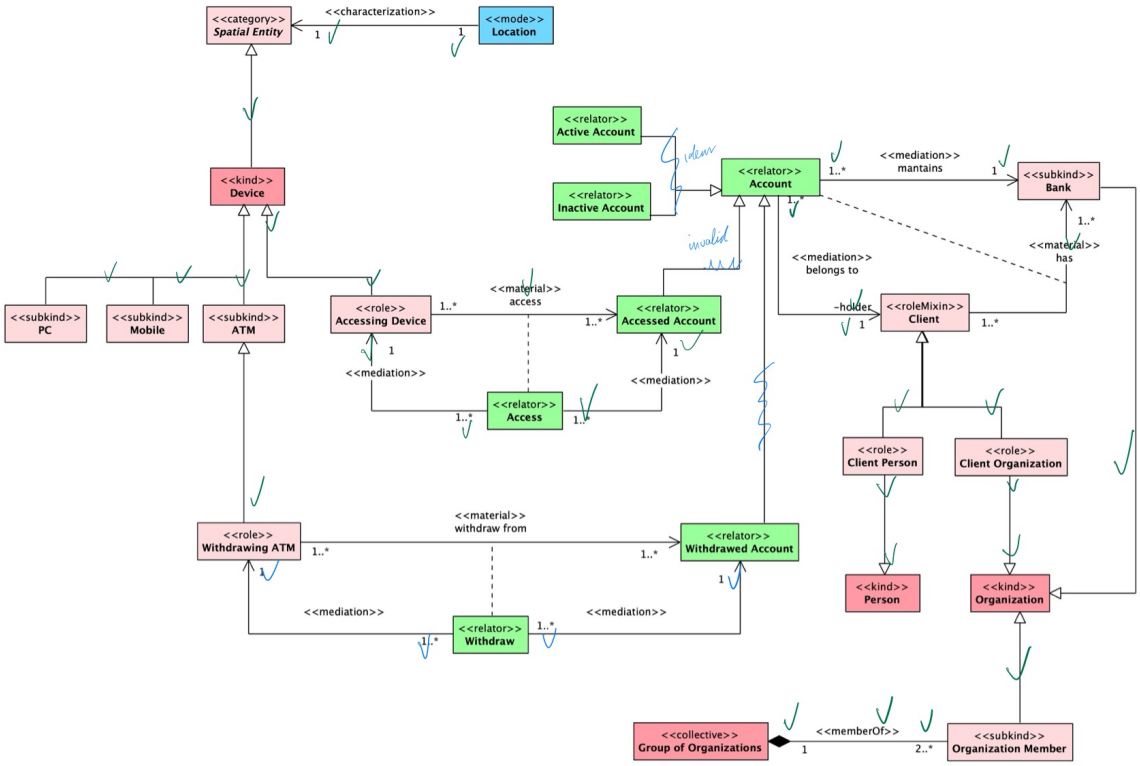


FIGURE C.11: Diagram of the bank-model model annotated for manual validation.

Validation checklist

OntoUML model: Barcelos 2013 Normative acts

		Passes	Fails	Does not apply
Classes in general				
	Are properties with types taken over into the generated class?	√		
Relations in general				
	For supported relation stereotypes, is a reference attribute present to the other class in both classes that are connected by the relation?	√		
	If the upperbound of the cardinality is > 1, is the reference attribute wrapped in the appropiate collection type?	√		
	Is the name of the reference attribute set to the type of the end (in case no relation end name is present), or to the relation end name in case one is present?			X
For identity providers				
	Is a class generated with the appropiate name?	√		
For generalizations (of roles)				
	Is a reference attribute pointing to the role present in the generic class, and vice versa?			X
	Is the reference attribute pointing towards the generic class marked as final?			X
	Are both reference attributes not wrapped in a collection type?			X
For generalizations (of phases)				
	Check if complex or simple scenario should apply			X
	For simple scenario:			
	Is an enumeration generated which contains all phases as literals?			X
	Is a reference attribute to this enumeration included in the generic class?			X
	For complex scenario:			
	Is an interface generated for the phase partition?			X
	Is a reference attribute generated in both the generic class and interface class?			X
	Is the reference attribute in the interface marked as final?			X
For generalizations (not connecting roles/phases to identitty providers)				
	If between to sortals: does the specific class extend the generic class?	√		

FIGURE C.12: First page of the filled-in checklist for the model barcelos2013normative-acts.

		Passes	Fails	Does not apply
	If between to non-sortals: does the specific interface extend the generic interface?			X
	If between sortal and non-sortal: does the specific class implement the generic interface?			X
For memberOf relations				
	Is the name of the reference attribute pointing to the member set to 'members' in case no relation end name is present? Otherwise, is that relation name used?	√		
For mediation relations				
	Is the reference attribute in the class corresponding to the relator marked as final?			X
	Is no reference attribute generated for a (possibly present) material relation?			X
For characterization relations				
	In case the moment end is a quality: are both involved reference attributes marked as final?			X
	In case the moment end is a mode: is only the reference attribute pointing towards the bearer marked as final?			X

* Based on this model, I decided to include char as a valid primitive type

* Includes a subcollectionOf relation
  ↳ (which is ignored)

* Includes non-primitive property types that don't have a datatype → so ignored

FIGURE C.13: Second page of the filled-in checklist for the model barcelos2013normative-acts.

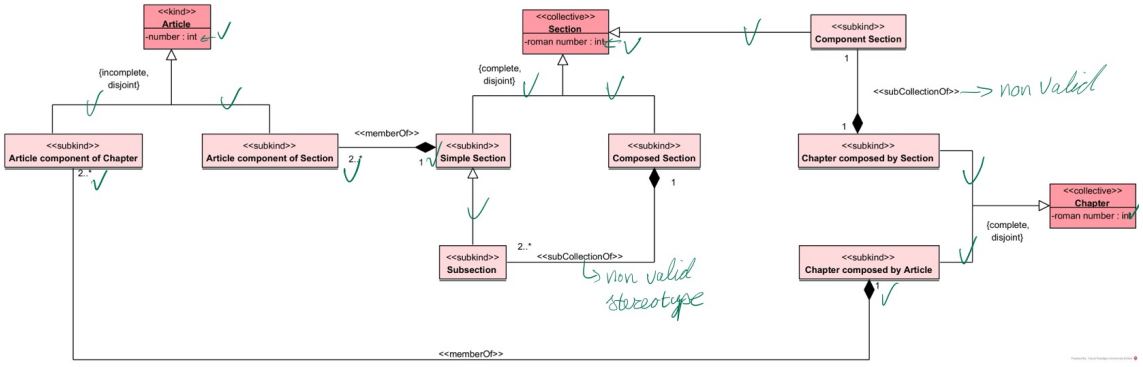


FIGURE C.14: First part of the barcelos2013normative-acts model annotated for manual validation.

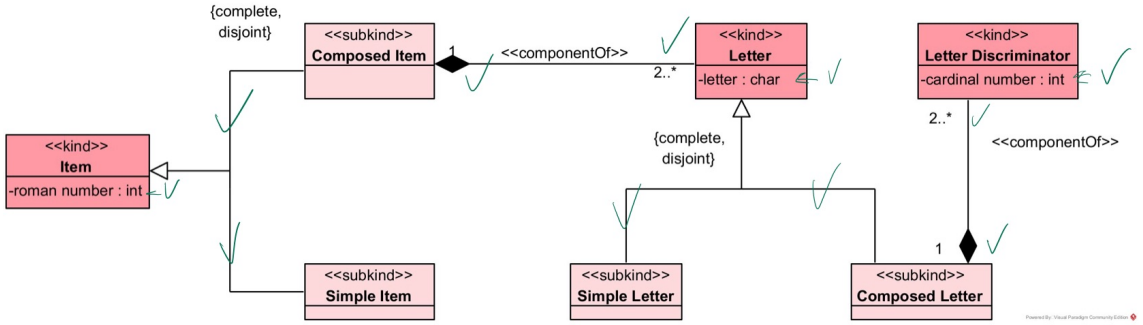


FIGURE C.15: Second part of the barcelos2013normative-acts model annotated for manual validation.

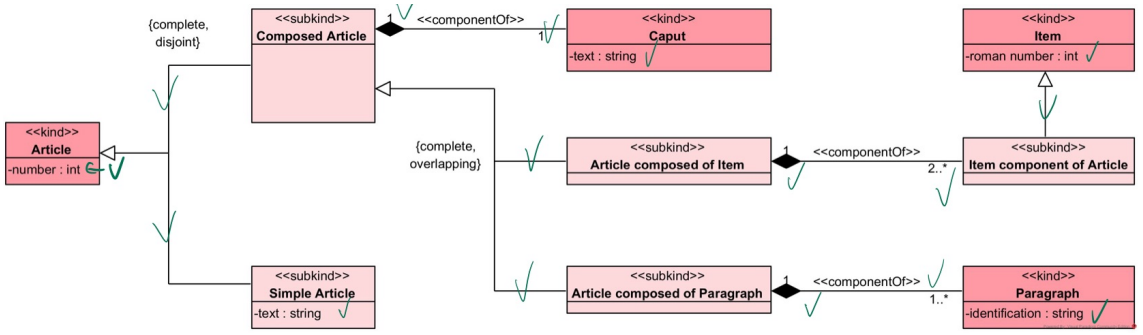


FIGURE C.16: Third part of the barcelos2013normative-acts model annotated for manual validation.

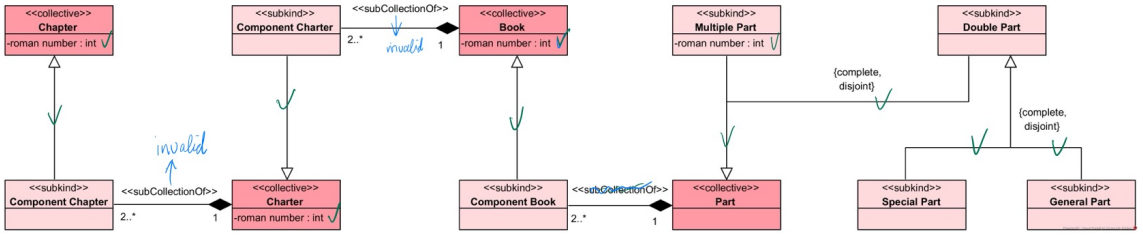


FIGURE C.17: Fourth part of the barcelos2013normative-acts model annotated for manual validation.

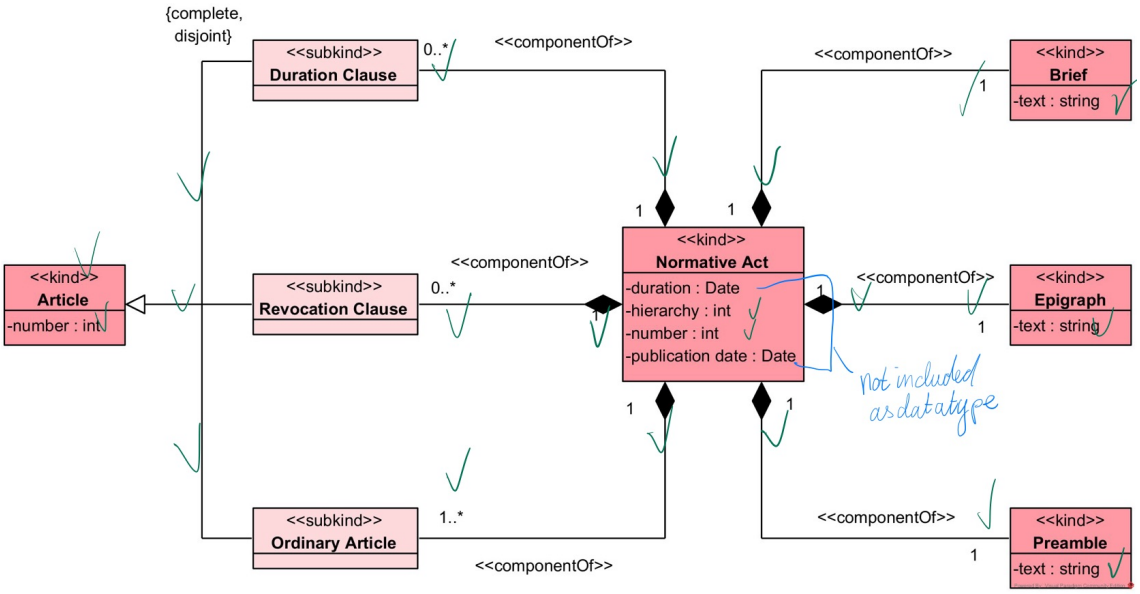


FIGURE C.18: Fifth part of the barcelos2013normative-acts model annotated for manual validation.

Validation checklist

OntoUML model: Barros 2020 programming

		Passes	Fails	Does not apply
Classes in general				
	Are properties with types taken over into the generated class?			X
Relations in general				
	For supported relation stereotypes, is a reference attribute present to the other class in both classes that are connected by the relation?			X
	If the upperbound of the cardinality is > 1, is the reference attribute wrapped in the appropiate collection type?			X
	Is the name of the reference attribute set to the type of the end (in case no relation end name is present), or to the relation end name in case one is present?			X
For identity providers				
	Is a class generated with the appropiate name?	✓		
For generalizations (of roles)				
	Is a reference attribute pointing to the role present in the generic class, and vice versa?			X
	Is the reference attribute pointing towards the generic class marked as final?			X
	Are both reference attributes not wrapped in a collection type?			X
For generalizations (of phases)				
	Check if complex or simple scenario should apply			X
	For simple scenario:			
	Is an enumeration generated which contains all phases as literals?			X
	Is a reference attribute to this enumeration included in the generic class?			X
	For complex scenario:			
	Is an interface generated for the phase partition?			X
	Is a reference attribute generated in both the generic class and interface class?			X
	Is the reference attribute in the interface marked as final?			X
For generalizations (not connecting roles/phases to identitty providers)				
	If between to sortals: does the specific class extend the generic class?	✓		

FIGURE C.19: First page of the filled-in checklist for the model barros2020programming.

		Passes	Fails	Does not apply
	If between to non-sortals: does the specific interface extend the generic interface?			X
	If between sortal and non-sortal: does the specific class implement the generic interface?			X
For memberOf relations				
	Is the name of the reference attribute pointing to the member set to 'members' in case no relation end name is present? Otherwise, is that relation name used?			X
For mediation relations				
	Is the reference attribute in the class corresponding to the relator marked as final?			X
	Is no reference attribute generated for a (possibly present) material relation?			X
For characterization relations				
	In case the moment end is a quality: are both involved reference attributes marked as final?			X
	In case the moment end is a mode: is only the reference attribute pointing towards the bearer marked as final?			X

* 'weird names' with special characters

FIGURE C.20: Second page of the filled-in checklist for the model barros2020programming.

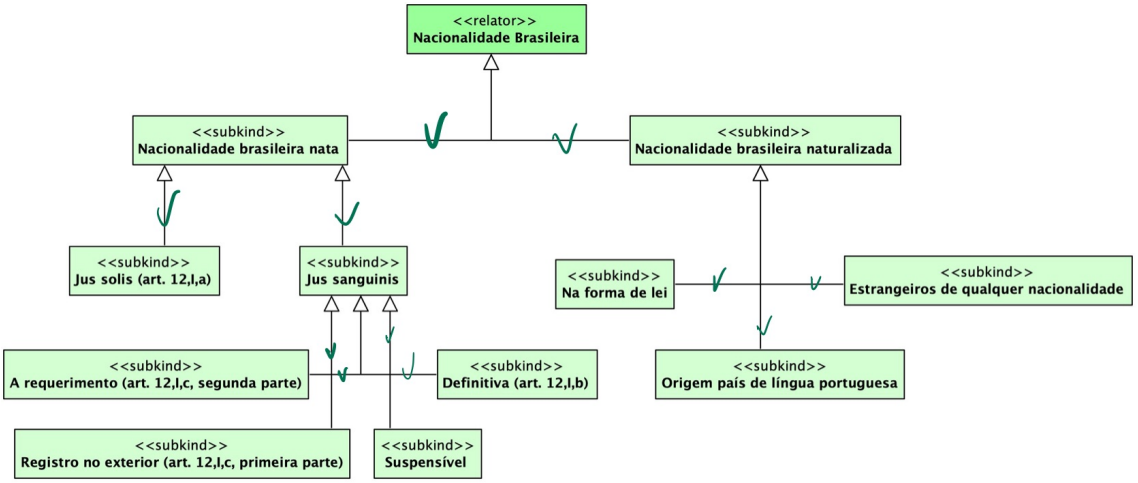


FIGURE C.21: Diagram of the barros2020programming model annotated for manual validation.

Appendix D

Online code repositories

The EMF project including the Ecore metamodel, ATL transformation, and Acceleo code generation are published at https://github.com/GuusVink/ontouml-java-generation. The other Python scripts used for both analysing the OntoUML model catalogue and executing and analysing the automated validation are published at https://github.com/GuusVink/GeneratingJavaFromOntoUML-auxiliary. Both repositories include README files with further information.

Appendix E

OntoUML limitations and ambiguities

This section contains references to the parts of the report that contain limitations or ambiguities found in OntoUML. These limitations/ambiguities vary in the degree of seriousness; some may be known and considered less important and others may be good to address in documentation.

Whether roles should always be associated with relators Section 3.3.2 describes that we assume that roles may be defined that are not associated with relators. However, some literature and documentation suggest that this should be the case.

Roles without relators might not be invalid, but rather incomplete models.

SubCollectionOf relation In the transformation of collectives as described in Section 3.3.1 and the implemented transformation as described in Section 5.3, we assume, based on OntoUML literature, that collectives may only have one *memberOf* relation.

Besides the *memberOf* relation, the online documentation lists another stereotype relating to collectives, namely *subCollectionOf* (also included in the VP plugin). However, this relation is not described in the OntoUML/UFO literature [18]. Even more so, we see consistency issues in the usage of this relation stereotype provided that our assumption of one *memberOf* relation is correct.

The *subCollectionOf* stereotype seems to imitate the *subQuantityOf* relation described in [17][1]. The *subCollectionOf* relation is bound to weak supplementation (according to the online documentation), meaning there should be at least two subcollections if one subcollection is present [17, 18]. This seems to contradict the fact that collectives may only have members that play the same role within the collective [18]; how else is the distinction between two subcollections relevant if they do not play the same role?

Considering that the *subCollectionOf* relation is not described in literature, as well as that it seems inconsistent with our *memberOf* assumption, we decided to ignore this relation for our transformation. We think it would be good to better specify these relations in an OntoUML specification as well as to add constraints relating these constructs to the VP plugin model validation.

Defining the values of qualities As described in Section 3.3.4, there seems to be no consistent method to define the possible values for qualities (e.g., a height quality

[1] Note that quantities are not covered by our transformation.

measured in centimeters or inches). The online documentation suggests a *structuration* relation although such a relation is not available in the VP plugin.

Dynamic aspects of modes In the transformation of modes as described in Section 3.3.4, we mark only one end of the characterization association final in the generated implementation model. Here we are not certain about the intended OntoUML/UFO semantics regarding whether instances of a mode may cease to exist, or conversely, whether they can be instantiated at any point in time.

Section 3.3.4 describes this ambiguity we experienced with a more concrete example. It could be valuable to specify this in an OntoUML specification.

Other recommendations Other recommendations for the OntoUML tools are discussed in Section 9.3, these apply more to the transformation we implemented in this research.

Bibliography

[1] Stefano Borgo, Antony Galton, and Oliver Kutz. Foundational ontologies in action. *Applied Ontology*, 17(1):1–16, 3 2022. doi:10.3233/ao-220265.

[2] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2017. doi:10.1007/978-3-031-02549-5.

[3] Roberto Carrareto. Separating ontological and informational concerns: a model-driven approach for conceptual modelling. Master's thesis, Federal University of Espírito Santo, 2012.

[4] Roberto Carraretto and João Paulo A. Almeida. Separating ontological and informational concerns: Towards a two-level model-driven approach. In *2012 IEEE 16th International Enterprise Distributed Object Computing Conference Workshops*, pages 29–37, 2012. doi:10.1109/EDOCW.2012.14.

[5] Thomas Derave, Tiago Prince Sales, Frederik Gailly, and Geert Poels. A method for ontology-driven minimum viable platform development. *Lecture Notes in Business Information Processing*, 450:253 – 266, 2022. Cited by: 1; All Open Access, Green Open Access. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85131327116&doi=10.1007%2f978-3-031-07475-2_17&partnerID=40&md5=80d4753c9763fde23e9cc55c4faa34e4, doi:10.1007/978-3-031-07475-2_17.

[6] Ricardo Falbo. Sabio: Systematic approach for building ontologies. *CEUR Workshop Proceedings*, 1301, 01 2014.

[7] Daniel Garijo and Maximiliano Osorio. Oba: An ontology-based framework for creating rest apis for knowledge graphs. In *The Semantic Web–ISWC 2020: 19th International Semantic Web Conference, Athens, Greece, November 2–6, 2020, Proceedings, Part II 19*, pages 48–64. Springer, 2020.

[8] Object Management Group. Model driven architecture (mda) rev. 2.0. Technical report, Object Management Group, 2014. URL: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01.

[9] Object Management Group. Unified modeling language (uml), v2.5.1. Technical report, Object Management Group, 2017. URL: https://www.omg.org/spec/UML/2.5.1/About-UML.

[10] Object Management Group. Meta object facility (mof) core specification, v2.5.1. Technical report, Object Management Group, 2019. URL: https://www.omg.org/spec/MOF/2.5.1.

[11] Thomas R Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.

[12] Nicola Guarino. Formal ontology in information systems. In *Proceedings of the First International Conference(FOIS'98), June 6-8, Trento, Italy*, 1998. URL: https://ci.nii.ac.jp/ncid/BA41968178.

[13] Gustavo L Guidoni, João Paulo A Almeida, and Giancarlo Guizzardi. Transformation of ontology-based conceptual models into relational schemas. In *Conceptual Modeling: 39th International Conference, ER 2020, Vienna, Austria, November 3–6, 2020, Proceedings 39*, pages 315–330. Springer, 2020.

[14] Gustavo L Guidoni, João Paulo A Almeida, and Giancarlo Guizzardi. Preserving conceptual model semantics in the forward engineering of relational schemas. *Frontiers in Computer Science*, 4:1020168, 2022.

[15] Giancarlo Guizzardi. *Ontological foundations for structural conceptual models*. Phd thesis - research ut, graduation ut, University of Twente, October 2005.

[16] Giancarlo Guizzardi. Agent roles, qua individuals and the counting problem. In Alessandro Garcia, Ricardo Choren, Carlos Lucena, Paolo Giorgini, Tom Holvoet, and Alexander Romanovsky, editors, *Software Engineering for Multi-Agent Systems IV*, pages 143–160, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[17] Giancarlo Guizzardi. On the representation of quantities and their parts in conceptual modeling. In *Frontiers in Artificial Intelligence and Applications*, volume 209, pages 103–116, 2010. doi:10.3233/978-1-60750-534-1-103.

[18] Giancarlo Guizzardi. *Representing Collectives and Their Members in UML Conceptual Models: An Ontological Analysis*, page 265–274. Springer Berlin Heidelberg, 2010. URL: http://dx.doi.org/10.1007/978-3-642-16385-2_33, doi:10.1007/978-3-642-16385-2_33.

[19] Giancarlo Guizzardi. Ontological patterns, anti-patterns and pattern languages for next-generation conceptual modeling. In *Conceptual Modeling: 33rd International Conference, ER 2014, Atlanta, GA, USA, October 27-29, 2014. Proceedings 33*, pages 13–27. Springer, 2014.

[20] Giancarlo Guizzardi, Alessander Botti Benevides, Claudenir M Fonseca, Daniele Porello, João Paulo A Almeida, and Tiago Prince Sales. Ufo: Unified foundational ontology. *Applied ontology*, 17(1):167–210, 2022.

[21] Giancarlo Guizzardi, Claudenir M Fonseca, João Paulo A Almeida, Tiago Prince Sales, Alessander Botti Benevides, and Daniele Porello. Types and taxonomic structures in conceptual modeling: a novel ontological theory and engineering support. *Data & Knowledge Engineering*, 134:101891, 2021.

[22] Hele-Mai Haav. A comparative study of approaches of ontology driven software development. *Informatica (Netherlands)*, 29(3):439 – 466, 2018. Cited by: 3. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85066799183&doi=10.15388%2fInformatica.2018.175&partnerID=40&md5=3f8675e33d5f11b800b2ae95d42836db, doi:10.15388/Informatica.2018.175.

[23] Hans-Jörg Happel and Stefan Seedorf. Applications of ontologies in software engineering. In *Proc. of Workshop on Sematic Web Enabled Software Engineering"(SWESE) on the ISWC*, pages 5–9. Citeseer, 2006.

[24] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F Patel-Schneider, and Sebastian Rudolph. OWL 2 Web Ontology Language Primer (Second Edition), 2012. URL: https://www.w3.org/TR/2012/REC-owl2-primer-20121211/.

[25] Thomas Hofweber. Logic and Ontology. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2023 edition, 2023.

[26] Olavo Holanda, Seiji Isotani, Ig Ibert Bittencourt, Endhe Elias, and Thyago Tenório. Joint: Java ontology integrated toolkit. *Expert Systems with Applications*, 40(16):6469 – 6477, 2013. Cited by: 15. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84879515691&doi=10.1016%2fj.eswa.2013.05.040&partnerID=40&md5=a7a30ca9b9c69a92cd7ec36f74ac60eb, doi:10.1016/j.eswa.2013.05.040.

[27] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, pages 128–138, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[28] C Maria Keet. *An introduction to ontology engineering*. University of Cape Town, 2018.

[29] Robert C Martin. *UML for Java programmers*. Prentice Hall PTR, 2003.

[30] Jeff Z Pan, Steffen Staab, Uwe Aßmann, Jürgen Ebert, and Yuting Zhao. *Ontology-driven software development*. Springer Science & Business Media, 2012.

[31] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[32] Robert Pergl, Tiago Prince Sales, and Zdeněk Rybola. Towards ontouml for software engineering: from domain ontology to implementation model. In *Model and Data Engineering: Third International Conference, MEDI 2013, Amantea, Italy, September 25-27, 2013. Proceedings 3*, pages 249–263. Springer, 2013.

[33] Michaël Verdonck, Frederik Gailly, Robert Pergl, Giancarlo Guizzardi, Beatriz Martins, and Oscar Pastor. Comparing traditional conceptual modeling with ontology-driven conceptual modeling: An empirical study. *Information Systems*, 81:92–103, 2019.

[34] Jan Vom Brocke, Alan Hevner, and Alexander Maedche. Introduction to design science research. *Design science research. Cases*, pages 1–13, 2020.

[35] Manuel Wimmer and Martina Seidl. On using uml profiles in atl transformations. In *Proceedings of the 1st International Workshop on Model Transformation with ATL (MtATL'09)*, 2009. URL: http://hdl.handle.net/20.500.12708/52722.

[36] Yongjie Zheng and Richard N Taylor. A classification and rationalization of model-based software development. *Software & Systems Modeling*, 12:669–678, 2013.