An Extended Concept of Delegation and its Implementation within a Modeling and Programming Language Architecture

Tony Clark[a], Ulrich Frank[*,b], Jens Gulden[c], Daniel Töpel[b]

[a] Aston University, Birmingham
[b] University Duisburg-Essen
[c] Utrecht University

Abstract. *Object-oriented modeling languages provide various concepts to express abstractions, which foster reuse and integrity. Among these concepts, generalization/specialization is of specific relevance. However, in many cases where generalization/specialization seems to be a natural choice, its use is likely to jeopardize a system's integrity. Delegation has for long been known as an alternative that allows preventing the accidental damage caused by the inproper use of specialization. Nevertheless, delegation is ignored by most tools for object-oriented modeling as well as by many textbooks, which may be caused by the fact that there is no unified conception of delegation, and that delegation is not supported by most object-oriented programming languages. This paper aims at a revival of delegation. To that end, the need for delegation is motivated by the analysis of counter-intuitive effects of specialization. Based on an extensive requirements analysis, a new, extended conception of delegation is presented. It allows for using delegation between classes on any (meta-) level and introduces the "delegation to class" pattern. Delegation in multi-level environments enables an additional reduction of redundancy and, hence, promotes integrity. The paper also presents design guidelines to foster the appropriate use of delegation. With respect to the implementation of delegation with object-oriented programming languages, two alternative flavours of delegation are analyzed. Finally, a prototypical implementation within a language engineering and execution environment does not only demonstrate the use of delegation in a modeling tool, but also its seamless implementation, by featuring a common representation of models and code at runtime.*

Keywords. Delegation • Inheritance • Specialization • Subclassing • Object-oriented • Conceptual Modeling • Software-Architecture

Communicated by Peter Fettke. Received 2021-03-08. Accepted on 2023-10-24.

1 Introduction

Reuse and adaptability of software systems are enabled through abstraction. On the one hand, abstracting invariant commonalities between system components enables the reuse of these common parts. On the other hand, abstracting away specific extensions or concretions allows for changing a system without creating side-effects on the invariant core, which includes fostering its reuse by adapting it to specific contexts. Among those concepts that foster abstraction, generalization and specialization are of specific relevance in software development. First, they correspond to abstraction concepts in everyday language we are all familiar with and should facilitate the design of systems that clearly reflect a certain domain of discourse. Second, they obviously address the need for reuse (through generalization) and adaptability (through

* Corresponding author.
E-mail. ulrich.frank@uni-due.de

specialization). Even though the potential benefits of generalization/specialization are undisputed, their use in software engineering has been accompanied by serious concerns. Some concerns reflect a skeptic rating of software developers' capabilities to use inheritance properly. Focussing on reuse, rather than on proper abstraction, that is, the conflation of specialization and inheritance may result in violating the Liskov substitution principle (Liskov and Wing 1994), which states that instances of classes can always replace instances of corresponding super-classes. In addition, there are doubts that software developers can easily identify specialization relationships that are invariant over time. As a consequence, inheritance hierarchies would have to be restructured, which implies substantial effort and risk. Other concerns are directly related to the concept of inheritance. Inheritance creates dependencies, which may jeopardise the reuse of classes in general, and their use in distributed environments in particular. Besides, inheritance creates invariant relationships that can hardly be changed after compilation, thus compromising dynamic adaptability.

Several proposals have been made to address the principal concerns with the concept of inheritance. Among those, two related approaches are of particular relevance. The concept of *role* was introduced long ago in data modeling (Bachman and Daya 1997) and in object-oriented software construction (Kappel and Schrefl 1991; Pernici 1990). On the one hand, roles are supposed to overcome conceptual limitations of classes or entity types, that is, to enable system models that more clearly correspond to the domain of discourse they represent. On the other hand, roles aim at adding flexibility to software by overcoming the static nature of inheritance relationships.

Delegation, which is closely related to the use of roles, has mainly been used in a more technical sense than with a conceptual intention. It emphasizes the message passing between two objects. An object A that represents another object B in a certain context delegates messages it receives and that are not part of its own interface, to object B. Delegation in this sense was not only proposed as a more powerful replacement of inheritance but was also used as a foundational abstraction mechanism of so-called prototype-based or class-less languages (Lieberman 1986). While the relationship between a role and a role filler object can be viewed as delegation, we shall see that not every delegation depends on the role metaphor.

Even though there has been a plethora of work on role-based software engineering and a considerable amount of work on delegation, we believe that a research gap remained for the following reasons:

- There is a considerable amount of publications on delegation. While the vast majority of these originate in the field of programming languages, there is a lack of studies on conceptual aspects of delegation and hardly any contribution that would account for both aspects. Fowler, for example, advocates delegation as an instrument to allow for selectively "inheriting" those operations of a class only that are actually needed, instead of overloading a class with various inherited operations that are actually not needed. (Fowler et al. 1999, p. 352). However, that addresses a problem, which is caused by (mis-) using inheritance for reuse purposes only.
- Related to the previous: peculiarities of inheritance have been a key motivation for the introduction of delegation. Authors usually point at harmful effects of inheritance but do not analyze the peculiarities of the semantics of object-oriented languages that cause these effects.
- There is no unified view of delegation or roles. In addition, the conceptual perspective and the implementation-oriented perspective on delegation, are not integrated, even though they are two sides of the same coin.
- In early discussions of the concept, delegation was sometimes distinguished from *forwarding*, e. g., (Kegel and Steimann 2008). While the distinction has substantial implications on the implementation of delegation and its potential

effects, it has not yet be analyzed how relevant it is from a conceptual perspective.

- Object-oriented programming languages hardly support delegation, which very likely contributed to the fact that there is only little use of delegation in practice.
- There is a lack of clear criteria that guide the proper use of delegation.

The paper is structured as follows Sect. 2 presents a critical review of inheritance that serves as the background for a subsequent discussion of questions concerning the conceptualization of delegation. It identifies the cause of counter-intuitive effects of specialization relationships in object-oriented models. This discussion will then be used to analyze requirements for delegation which leads to the concept of delegation proposed in this paper. It emphasizes a conceptual viewpoint, but accounts for programming languages, too. It also accounts for a conceptual comparison of delegation and forwarding. In addition, delegation will be extended to allow for its use in multi-level modeling. It enables delegation between classes on any (meta) level as well as delegation across levels, that is, "delegation to class", which allows objects to transparently access operations offered by their class and, thus, the state of their class. Delegation in multi-level environments enables an additional reduction of redundancy and, hence, promotes integrity. A set of modeling guidelines for using delegation properly is developed in Sect. 3. Sect. 4 presents a language architecture that we propose as being suitable for exploring different implementations of delegation. The language architecture is implemented by the tool XModeler^{ML}, a multi-level modeling and execution environment (for a description of the recent implementation see Frank et al. 2022). It is based on the language engineering environment XModeler (Clark et al. 2008a,b and has been developed within the project "Language Engineering for Multi-Level Modeling" (https://le4mm.org/, Frank and Clark 2023). The language technology provided by the XModeler has been used to realise the two alternative implementations of the delegation presented in Sect. 5. The approach is compared to existing work in Sect. 6, before final reflections on conclusions and future work in Sect. 7 complete the article.

Note that this paper may be perceived as unusual since it combines conceptual and methodical considerations of delegation with detailed, rather technical discussions of implementation aspects. However, all these aspects need to be accounted for, if one wants to enable the convenient and efficient use of delegation. If delegation remained a concept only without regard for its implementation, software development would hardly benefit from it. This is the case, too, if no guidelines for its proper use are provided. Those readers who are not interested in technical details may skip Sect. 5 and parts of Sect. 4 without the risk of essentially diluting the entire picture.

2 Delegation: Motivation and Conceptualization

Our work on the analysis and development of a flexible language-based approach to delegation is particularly motivated by problems caused by specialization in object-oriented software development. Note that the majority of publications on delegation compare it against inheritance, where inheritance is not specifically treated as a conceptual term, bur rather as an instrument that enables reuse. A frequently used example is that of the class `Stack` that inherits from the class `Vector`, cf., e. g. Fowler et al. (1999) and Kegel and Steimann (2008). That leads, among other things, to the problem that `Stack` does not only inherit useful operations but also operations that do not make sense for stacks, like "insertElement ..". However, this kind of inheritance is suspicious anyway, because it does not make sense from a conceptual perspective, since a stack does not qualify as a vector.

Different from inheritance, specialization should not violate the substitutability constraint (Liskov and Wing 1994), that is, wherever an instance of a superclass is required, an instance

of one of its subclasses can be used as a valid replacement. In the following, we will use the term "specialization" in those cases where the substitutability constraint should be satisfied. In cases where this is not essential, we speak of "inheritance". This section presents a critical review of specialization that will provide the background for a first discussion of the notion of delegation. To sharpen this notion, we will look at potential use cases for delegation and corresponding requirements to finally present a comprehensive definition of delegation from a conceptual perspective.

2.1 A Critical Review of Specialization

Generalization is a powerful abstraction. It promotes integrity by capturing commonalities shared by a set of classes and, thus, supports avoiding conceptual redundancy. It also supports reuse and adaptability through specialization, which allows for monotonic extensions. Nevertheless, the use of specialization in object-oriented systems can cause problems for the unwary because of its counter-intuitive semantics. To illustrate the problem, we look at a simple example of specialization in natural language: a student is a person. In logic, this proposition corresponds to

$$\forall x \; student(x) \implies person(x)$$

In other words: every statement that holds for the general concept, must hold for the specialized concept, too. That applies to specialization in object-oriented systems, too, since a specialized class inherits all properties from its superclass. The example class diagram in Fig. 1 shows a corresponding model. It might be part of an object model that was designed for a software system to manage human resources at a university. Among other objects, the system should represent students and employees. Like in logic, such a conceptualization seems to be an obvious choice. However, different from logic, it may produce serious anomalies.

Sometimes, students turn into employees after they obtain a degree, that is, after they stop being students. There are also cases, where a student is

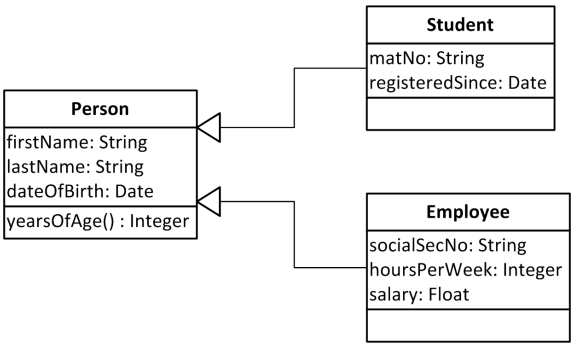

Figure 1: Intuitive use of inheritance

employed as a student assistant. In any case, a system that implements the model in Fig. 1 would be threatened by redundancy. Every instance of a subclass would have to redundantly represent the state of the corresponding instance of the superclass (see Fig. 2). This would be especially dangerous, because without further measures an object of a subclass would not know the corresponding object of the superclass. If an instance of a subclass is to be merged into an instance of another subclass, e. g., when a student turns into an employee after graduation, it gets even more dangerous. The relevant state of the employee object has to be copied either from the corresponding person object or the corresponding student object. Subsequently, the student object has to be deleted, which includes accounting for and possibly redirecting references to and from other objects.

At first sight, this effect of specialization is confusing because it seems counter-intuitive. In fact, it does not occur in logic or set theory. An element that satisfies the constraints defined for the concept `Student`, is an element of the set `student` *and* of the set `Person`. If in addition, it satisfies the constraints defined for the concept `Employee`, it would become an element of the set `Employee`, too. If the specific properties that characterize a student are removed from an element, it will no longer be part of the set `Student`, but remain in the set `Person`.

The problem is caused by the specific concept of class in object-oriented languages, where every object is an instance of one and only one class.

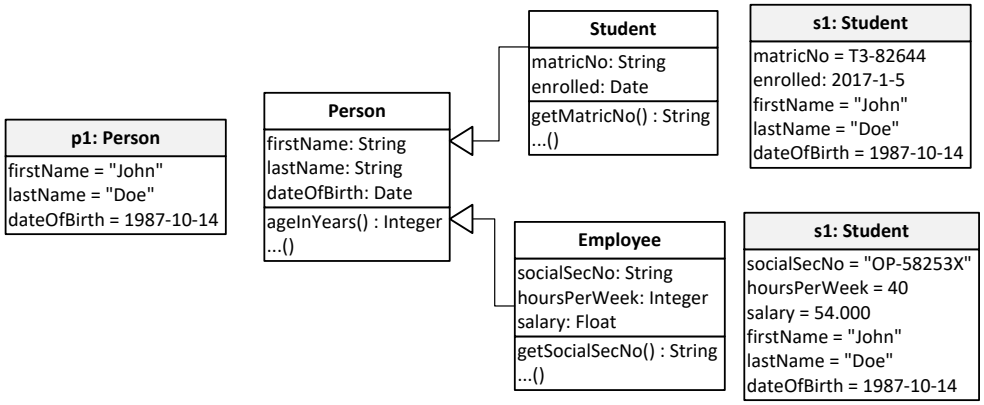

Figure 2: Specialization leading to redundancy

A class is defined *intensionally* through a set of properties that is characteristic of all its possible instances. A specialized class inherits these properties and extends them with additional ones. As a consequence, an object of a specialized class qualifies as a proper object in all cases where an object of the corresponding superclass is required. Therefore, it satisfies the substitutability constraint. Nevertheless, it has an implication that does not fit our natural conception of specialization. An object of a specialized class is not an instance of the superclass at the same time, which corresponds to a seemingly bizarre statement like "a student does not belong to the set of persons", or, more drastically, "a student is not a person". This notion of a class is comparable to a physical template that is used to punch out objects of a certain shape; applying a further template to the object would result in destroying its original shape. Hence, an object that represents a student cannot represent a person (an instance of the class `Person`) simultaneously. Instead, the state of an object that represents a student would have to redundantly replicate the state of a person.

This is different in logic, where an object can be subsumed under one to many classes reflecting an extensional, set-oriented view of class, i. e., a class is conceptualized as a set of objects that share the same properties. Hence, if a class `Student` is subordinated under a class `Person`, every element of `Student` becomes an element of `Person` simultaneously. As a consequence, an object representing a student shares values representing properties like `firstname` and `lastname` with an object that represents a corresponding person. The notion of class in natural language corresponds to the one used in logic, which explains why the dangerous effects of specialization in object-oriented systems may be perceived as counter-intuitive. Despite the extensive analysis of delegation, we are not aware of any publication, except for Frank (2000), that pointed at this essential cause of the counter-intuitive effects of inheritance.

One might assume that experienced software engineers are aware of the problem and know measures to circumvent it. We hope so. However, it seems that the counter-intuitive notion of class in object-oriented systems does not only represent a trap for novice programmers (Tempero et al. 2013). A random survey of 15 textbooks on object-oriented software development revealed 10 books with example uses of specialization that could lead to the anomalies described above. In none of those cases, the authors pointed to the potential problem (Taylor 1990, p. 23, Coad and Yourdon 1991, p. 46, Rumbaugh 1991, p. 62, Embley et al. 1992, p. 100, Yourdon 1994, p. 242, Ayesh 2002, p. 21, Ambler 2004, p. 39, Ambler 2005, p. 66, Balzert 2011, p. 120, Rau 2016, p. 105).

Against this background, it may seem sensible to modify the semantics of *class* in object-oriented

languages, for example by allowing objects to have multiple disjoint classes. Whilst this might be an interesting research theme, virtually all practical object-oriented software engineering tools are based on the notion of a single, or in the case of inheritance, homogeneous, type system since there is a close correspondence between the type of a data element and its representation in memory.

2.2 Use Cases: Facets of Delegation

Delegation has been discussed for more than two decades. While it seems that there are no objections against the claim "Delegation is a powerful programming tool." (Szyperski et al. 2002, p. 138), there is no unified notion of delegation. In the area of programming languages, delegation is often restricted to dynamic aspects, that is, passing messages received by an object to another object (Kegel and Steimann 2008; Szyperski et al. 2002). Some authors introduce delegation as a core abstraction of programming languages. This is especially the case for prototype-based or classless languages, where delegation serves as a substitute for inheritance. However, while Lieberman (1986) regards delegation to be more powerful than inheritance, Stein comes to the conclusion that "delegation is inheritance" (Stein 1987) (we look closer at the subtleties of these conceptualizations in Sect. 3.1 and 6.2). A further stream of research with a more conceptual focus makes use of the term "role" to denote a relationship between objects that allows for a higher degree of flexibility than inheritance (e. g. Bachman and Daya 1997, Steimann 2000b, Halpin 1995).

We suggest using the term delegation for both a conceptual view of a relationship between objects and a more technical view of the dynamic aspects of this relationship. Both views have in common that delegation denotes a directed relationship between an object (the "delegatee") and a context-specific extension or representative (the "delegator") of that object. The corresponding classes are referred to as "delegatee class" and "delegator class", respectively. Furthermore, different from inheritance, delegation fosters flexibility, that is, a delegator object can be removed or replaced during runtime. At the core of the technical view on delegation is the idea that the delegator object dispatches messages to a corresponding delegatee object if a message it receives is not implemented or not included in its own interface. In the ideal case, the message dispatch is transparent to the object that sends a message to a delegator object. Fig. 3 illustrates this idea. The circle marks the class that represents the delegator. As a consequence, the delegator object "inherits" not only the interface but also the state of a corresponding delegatee object. Note, that at this point we do neither account for the question of whether the delegator object has the same identity as the delegatee object (Szyperski et al. 2002, p. 110, Bettini et al. 2003), nor for the kind of programming language, that is, whether it is based on dynamic or static typing. We will consider both aspects later.

Before we get back to implementation issues, we will first develop an elaborate conceptual view of delegation. A conceptual view requires creating a correspondence between delegation and relationships we use in natural language. Only then, it is possible to appropriately use delegation for the representation of a certain domain of discourse, because delegation is not a common relationship in natural language, nor do the terms "delegatee" or "delegator" clearly indicate what kind of objects they might represent. For this purpose, we will look at a few selected use cases of the delegation mechanism to analyse how they could be conceptualized and whether they should be included in the conceptual view of delegation.

Roles: The prevalent approach to conceptualizing delegation relationships is the introduction of roles. The concept of role is popular in natural language. An actor can play different roles at a time and may change the roles played during his / her lifetime. Accordingly, one object in a delegation relationship would represent a role, the other one the role filler, e. g., an actor. Role is a conceptual refinement of the delegator, role-filler a conceptual refinement of delegatee. A delegation association would then connect a class that represents role fillers and a class that represents corresponding

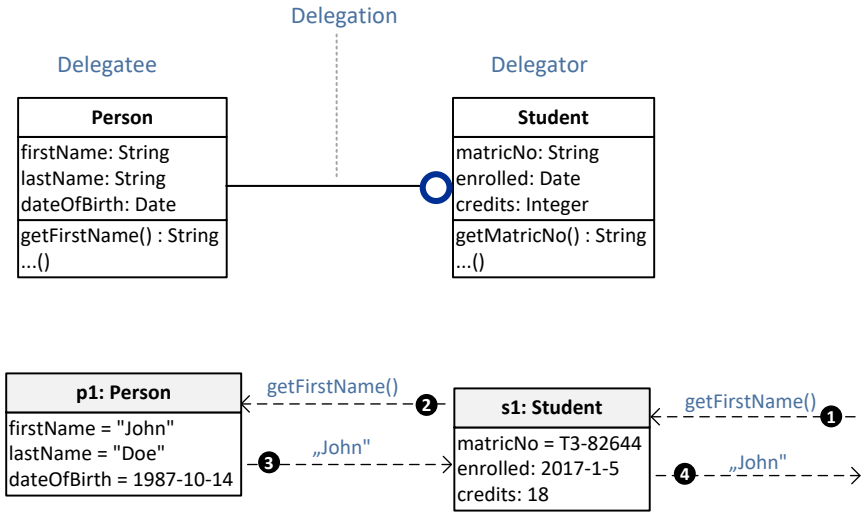

Figure 3: Illustration of dynamic aspects of delegation

roles. The delegatee and delegator in Fig. 3 are good examples of role and role-filler. Roles are of outstanding relevance for the conceptualization of delegation. There are many cases where the use of roles allows for avoiding the negative effects of specialization. At the same time, using roles as a conceptualization seems natural, that is a directly convincing choice.

Projection or "selective" delegation: Usually, delegation implies that every method call, a delegator object cannot respond to with one of its own operations, is delegated to the corresponding delegatee object. This abstraction has two advantages. First, the delegation of method calls does not have to be defined explicitly for all operations of a delegatee object. Second, related to the first, modifying the interfaces of the delegatee object's class does not require corresponding adaptations of the delegator class. However, there may be cases where only explicitly selected operations of the delegatee class should be subject of delegating corresponding messages to a delegator object. Fowler recommends this approach to refactor applications of inheritance that lead to inheriting operations that do not fit the semantics of the inheriting class. For illustration purposes, he uses the already mentioned example of the classes `Stack` and `Vector`. Hence, the object of `Vector` would fulfill the only purpose of serving as a partial projection of the object of `Stack`. In addition, this kind of selective delegation seems to promote flexibility: "I can delegate to many different classes for different reasons." (Fowler et al. 1999, p. 399)

It can make sense to explicitly delegate certain method calls for refactoring purposes to heal the potential damage caused by inheritance. Similarly, there may be reasons to forward method calls to objects of different classes. However, this kind of "hard wired" delegation should be done with extreme care. This is mainly for the reason that there is no clear conceptual relationship between the delegator and the delegatee. It corresponds to applying inheritance only for the purpose of reusing certain operations and is reduced to the notion of explicitly forwarding selected messages in a static sender-receiver relationship. Abstractions provided by the conceptual understanding of delegation elaborated in this article are not taken advantage of.

Polymorphic Delegation: There are cases where a delegator class may be associated with more than one delegatee class. For example, the delegatee class `Interpreter` could be associated with a delegator class `Person` if a human is acting as a language interpreter, or with a delegator class

`Translator` in case a software system is used. A further example would be customers of a firm that can either be consumers or organizations. In both cases, delegation would be an especially useful choice, if not only the delegator object assigned to a delegatee could change during the lifetime of a program, but also the class of the delegator object. Also, in both cases, the role metaphor is the proper choice for the conceptualization of the relationship: interpreter can be regarded as a role that is filled by a person or a software system. Similarly, customer can be seen as a role of a consumer (person) or of an organization. However, replacing the class of a role filler during the runtime of a program will cause a problem, if the interfaces of both classes are different. Therefore, multiple delegatee classes with a delegation association need to have a common set of methods. The example in Fig. 4 illustrates that the commonalities between two delegatee classes, each of which may make perfect sense, do not have to come natural. Therefore, the benefits of polymorphic delegation have to be compared to the drawbacks created by all too artificially constructed commonalities. Polymorphic delegation corresponds to polymorphism through inheritance and directly reflects the idea of "inclusion polymorphism": "In inclusion polymorphism an object can be viewed as belonging to many different classes which need not be disjoint, i. e., there may be inclusion of classes." (Cardelli and Wegner 1985, p. 5) While this kind of inclusion in a strict sense is not possible in object-oriented systems, delegation at least allows getting close to it, because a delegator object has access to its delegatee object's state. Steimann emphasizes a different view on polymorphism through delegation: "An instance is polymorphic if it can play different roles ..." (Steimann 2000b, p. 103). In that sense, delegation is polymorphic per se.

Delegation Chain: A delegator object may serve as a delegatee object for a further delegator object. The example in Fig. 5 shows a delegation chain that consists conceptually of (hybrid) delegators and delegatees. Delegation chains are also suited for the representation of configurations that include variants on different levels of composition. For example, a PC as a core product is offered in two variants, one with a hard disk drive, the other one with a solid state drive. Further variants of the PC could consists of variants of these two drive types. While delegation chains are suited to increase a system's flexibility, their design demands for a thorough analysis of possible future changes of requirements (see guidelines in Sect. 3.2).

2.2.1 Delegation versus Forwarding

Authors that discuss issues related to the implementation of delegation, often emphasize that delegation should not be confused with forwarding, e. g. Fowler et al. (1999) and Kegel and Steimann (2008). The distinction relates to a subtle issue, that is, the question whether self (or any other corresponding keyword) within an operation that is executed by a delegatee object as a consequence of a message delegated (or forwarded) to it by one of its delegator objects should refer to the delegatee object or to the delegator object. It is not obvious how to judge this proposed difference between delegation and forwarding from a conceptual perspective. The examples shown in Fig. 6 demonstrate the effects that may be caused by the two different implementations. If the delegator object receives a message like `age()` that is not included in its interface, it will delegate (or forward) it to the delegatee object it is linked to. Assume that, for some reason, the implementation of `age()` in the class `Person` refers to a further operation in the delegatee object, e. g., `ageInYears()`. In this case, it would be appropriate, if self represented the delegatee object. If the interfaces of both classes include the same signature, e. g.`getID()`, and, for some reason, an operation executed by a delegatee object would refer to `getID()` via `self`, that would likely produce an inconsistent result, because it would use the id of a person, where the id of the delegator object would be required. To cope with this case, Kegel and Steimann suggest what they call "reverse delegation" (Kegel and Steimann 2008, p. 434). The delegatee object

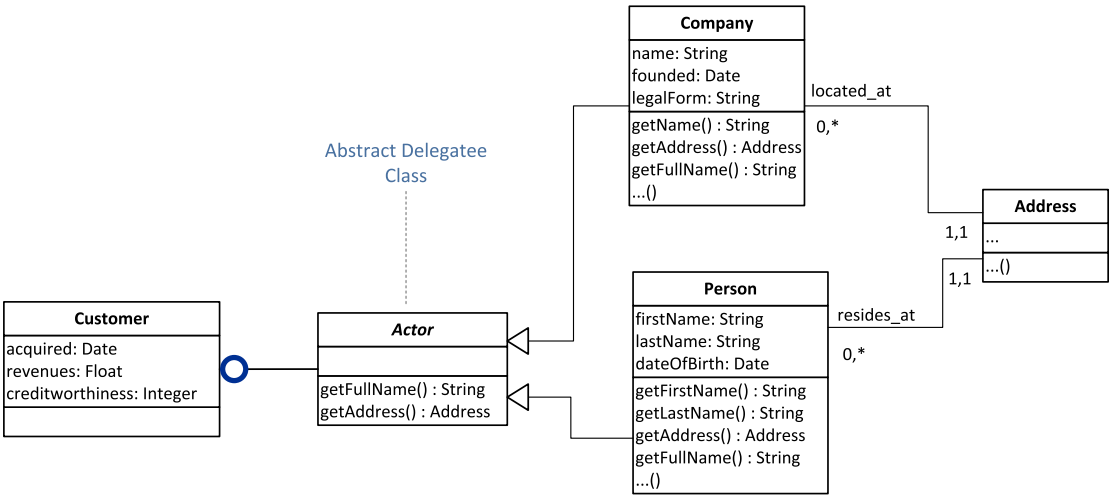

Figure 4: Illustration of polymorphic delegation

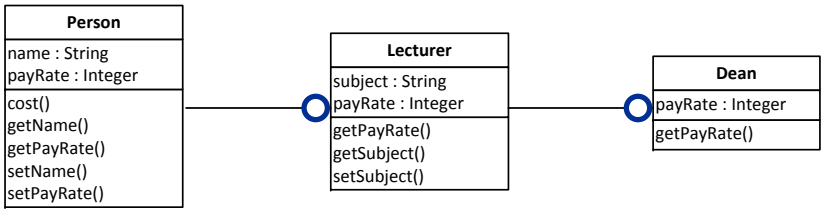

Figure 5: Example of a delegation chain

would then "explicitly dispatch" to the corresponding delegator object. That would also be satisfactory for a case, where a message sent to `self` by an operation executed by a delegatee object could not be handled by the corresponding delegator object, as it is the case with `ageInYears()` in the example. In that case, the delegator object would dispatch the message back to the delegatee object. Apart from the challenge to decide when to call the method provided by the delegatee object itself and when to dispatch it back to the delegator object, there is a further issue that makes the implemenation even more demanding. It relates to the example of the method `printAsString()`, which is offered by both the delegator and the delegatee object. If the implementation of the operation in the class `Student` calls the respective operation in the corresponding delegator object, as indicated in Fig. 6, the operation would not terminate. Similar issues arise in multiple inheritance where name resolution can become ambiguous. Languages resolve such cases either by clearly defining one, perhaps arbitrary, implementation choice, or by providing language features so that the modeller or programmer is required to explicitly link the definiens with the definiendum. Our approach adopts the former whilst leaving room for future extensions in terms of the latter, so that, for example, we must be careful to interpret `self` in terms of a default non-delegating object otherwise `self.printAsString()` would cause infinite regress.

Apparently, the implementation of delegation with `self` in the delegatee object referring to the delegator object is challenging. However, that does not have to be a serious problem. From a conceptual perspective, delegation and forwarding are equivalent as long as one precondition is satisfied. Every operation in a delegator class that refers to operations via self must not do that in case the respective signature is offered both by the delegator and the delegatee class. While this may

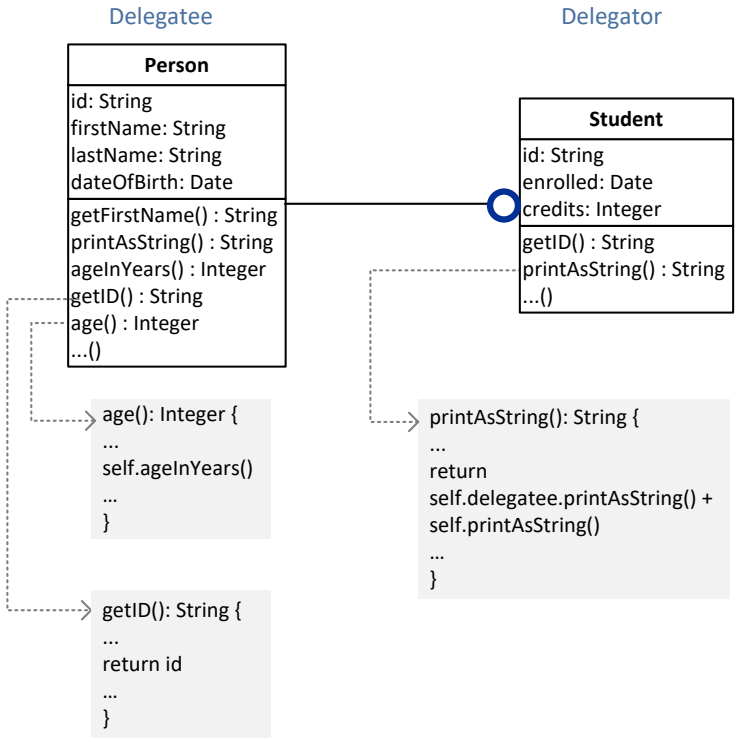

Figure 6: Illustrating the effect of `self` *representing either the delegator or the delegatee object*

seem as a critical precondition, our long standing experience with using delegation does not include a single case where `self` within an operation of a delegatee object would have had to refer to the delegator object. We will look at the implementation of both, forwarding and delegation in later sections (5.1, 5.2).

2.2.2 Multi-Level Delegation

Different from traditional approaches to conceptual modeling that are based on MOF-like language architectures, multi-level modeling (Atkinson and Kühne 2001, 2008; Frank 2014; Neumayr et al. 2009) allows for an arbitrary number of classification levels. Every class, no matter on what level it is, is an object at the same time, which means it may have a state and can execute operations. The additional abstraction enabled by multi-level language architectures allows avoiding conceptual redundancy, and, thus, promote reuse and integrity. The analysis of multi-level models indicates that delegation serves as a useful abstraction to enable transparent dispatch of messages between classes and between an objects and their classes (Frank 2018).

Variants: There is no unified definition of a variant. However, there seem to be two key characteristics. First, a variant may extend a virtual or real core artefact by additional properties. For example, a variant of a scanner may come with an additional document feeder. Second, it may change or replace properties of the core, e. g., a variant has a different weight. Apart from the variations, a variant does not only share its properties, but also specific property values with the core. Fig. 7 shows how delegation can be used to represent the relationship between a variant and a core artefact. The specification of a class that represents a variant would have to include only those properties of the variant that are supplementary or that have a different value. The benefit of delegation is obvious: if the technical specification of the core is modified such that certain properties

are characterized by new values, the variants do not have to be updated. Note that it is conceivable for a core artefact to be virtual only. If `Scanner` is used to represent commonalities of a family of scanner types, but does not represent a type of its own, there will be no real object corresponding to its instance.

Polymorphic delegation can be applied to variants, too. If, for example, the same document feeder type is used to form variants of various scanner types, one variant class could be associated with various core product classes.

Delegation to Class: The variant and the core product in the example in Fig. 7 actually represent product types. Therefore, it may be required to instantiate them into objects that represent particular physical devices, each of which may carry a unique serial number. In the traditional paradigm which allows for objects on M0 only, this kind of instantiation/classification could not be represented without some kind of overloading. By contrast, multi-level modelling and programming ((Atkinson and Kühne 2001; Clark et al. 2014; Frank 2014)) allows for an arbitrary number of classification levels, where every class is an object at the same time. In other words: classes may have a state. This does not only add flexibility, but it also contributes to integrity, because it allows for reducing redundancy. If, for example, all instances of a class need to have the same value of a certain property (for example, all printers of a certain model have the same sales prices), this value can be stored with the class, instead of storing it with every single instance. In that case, any particular exemplar (e. g., every particular printer) is characterized by this value. Therefore, it makes sense to hand over the message that requests this value, directly to the object representing the exemplar.

As the example in Fig. 8 shows, delegation would enable the object to properly respond to such a request without storing the corresponding value redundantly. To that end, the message would have to be delegated to the object's class. If the method is not included in the interface of the class, it could be further delegated. The model in Fig. 8 was created with the multi-level modelling language FMML^x (Frank 2014). Other multi-level modelling languages such as (Atkinson and Gerbig 2016; Neumayr et al. 2014) allow for similar representations.

Like projection, delegation to class lacks the flexibility that is regarded as an important aspect of delegation, since it represents an instantiation relationship that is invariant through the lifetime of an object. However, unlike projection, there is a conceptual relationship between delegator and delegatee that makes sense. It is very common to represent values that apply to all instances of a class with the class itself. Examples comprise technical specifications, biological taxonomies, and many others. Furthermore, it is conceivable that new properties and corresponding values are added to a class during the lifetime of its instances. However, even though delegation to class can be useful, it comes with serious challenges. Not every class includes representations of common values shared by its instances. If an object that serves as a delegator in a delegation relationship with an object on the same classification level receives a message that is not in its protocol, it needs to be decided whether the message should be dispatched first to its class or to the other delegatee. Apart from that, delegation to class does not allow for polymorphic delegation, because an object has one, and only one class. Delegation hierarchies are conceivable in a multi-level language architecture. Take, for example, a conceptual hierarchy like species, genus, family, order, etc. Each concept that is assigned to one of these hierarchies may define values that are representative for all its elements.

The above examples give an overview of how delegation might be conceptualized. They also indicate how to assess the different kinds of conceptualization with respect to their contribution to flexibility, their conceptual validity and their suitability for polymorphic delegation as well as delegation chains. Tab. 1 summarizes this intermediate result.

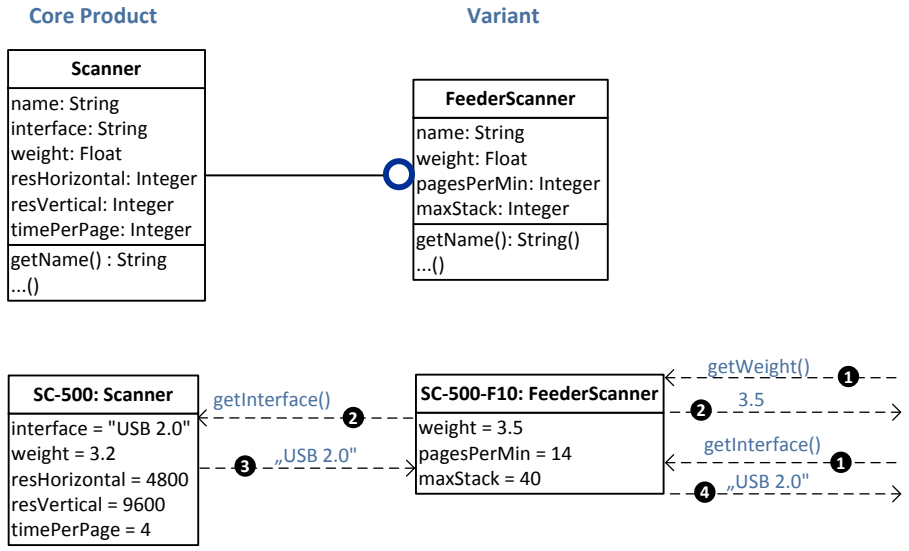

Figure 7: Use of delegation to represent variants

Levels	Delegatee	Delegator	*Flexibility*	*Validity*	*Polymorphic*	*Chains*
single-level	Role Filler	Role	high	convincing	✓	✓
	Projected Class	Class	high	inappropriate	×	✓
multi-level	Core Artefact	Variant	high	convincing	✓	✓
	Instance	Class	low	reasonable	×	✓

Table 1: Conceptual flavors of delegation and their preliminary assessment

2.3 Analysis of Requirements

The examples need to be analyzed in more detail in order to develop more elaborate requirements. The following list includes the requirements we identified and their relation to the above use cases. Static requirements toward language elements for defining delegation and the way they can be applied are marked as "SR". "MR" refers to applying delegation in multi-level language environments. Dynamic requirements, which refer to the expected behavior related to delegation, are marked as "DR". A further category, which relates to corresponding modelling and software development tools, is marked as "TR". Some of the requirements reflect extensions we made to previous conceptions of delegation. They are marked with "extension". Those requirements that relate to optional features are marked with "optional". Note that requirements are not always expressed in an intentional form, but also as assertions. It is not possible to guarantee the completeness of requirements. To ensure that they cover a wide range of relevant aspects, we compare them to corresponding features of role-based modelling approaches that were identified by Steimann in the most comprehensive literature study on this topic (Steimann 2000b). In the following we refer to classes that specify delegators as delegator classes, to corresponding objects as delegator objects or delegators. Classes that represent delegatee are referred to as delegatee classes and their instances as delegatee object or delegatee respectively.

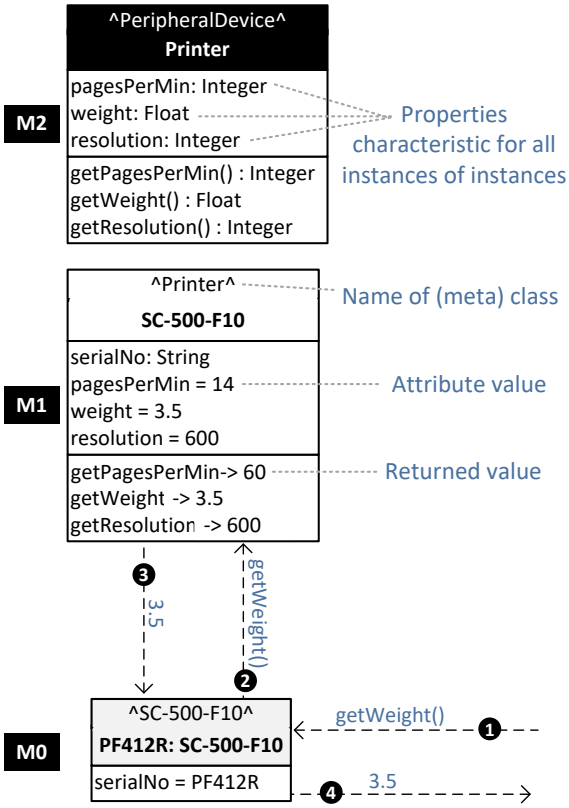

Figure 8: Delegation to class

2.3.1 Static requirements

SR1 A delegatee class is associated to one to many delegator classes. *Rationale*: This property is at the core of the abstraction enabled by delegation. If no further constraints apply, this property implies that a delegatee object can be linked to more than one delegator object simultaneously (corresponds to feature #3 in Steimann (2000b)).

SR2 A delegatee object can be linked to more than one delegator object of the same class simultaneously. *Rationale*: In many cases the number of delegator objects of the same class is restricted to one. However, there are cases where a delegatee is linked to more than one delegator object of the same class. A person may be employed at different organizations at the same time. Variants linked to a core artefact may all be of the same class (corresponds to feature #4 in Steimann (2000b))

SR3 *(optional)* It should be possible to associate a delegator class with more than one delegatee classes. *Rationale*: The ability to abstract the class of possible delegatee objects away is suited to clearly increase the flexibility of a system, since it would allow for replacing a delegatee object with another delegatee object of a different class at runtime. However, allowing for such a modification without any restrictions creates a substantial risk. If the new delegatee object's interface is different from the one of the replaced object, messages that worked previously would fail. Such a side-effect is hardly acceptable. Therefore, it should be possible to associate more than one delegatee class only, if the delegatee classes share an abstract superclass that defines a common interface (polymorphic delegation, see Fig. 4). This feature is mentioned in #7 in Steimann (2000b).

SR4 Delegatee and delegator are objects with an own identity, that is, they do not share the same identity. *Rationale*: This requirement is different from most concepts of roles found in literature, where delegators are regarded as part of the delegatee object (Steimann 2000b, p. 87). There are various reasons why we decided for this requirement. The first one is of ontological nature. The existence of an object that serves as a delegatee is independent of the delegators linked to it. At the same time, a particular delegator object may have an existence of its own, too. For example, there is only one role of a dean in a university department and it is filled with different persons over time. The lack of existential dependency becomes even clearer with the additional conceptualization of delegation we introduced above: a variant of a product does not stop to exist, if the core product is removed. However, there are delegation relationships, where a delegator object is existentially dependent on its delegatee object, e. g., an object representing a student. Therefore, this ontological argument is only partially valid.

The second reason is ontological, too. If delegatee and delegator share the same identity, the delegatee would include the delegator's properties as essential own properties. For example, a person would essentially include the properties of a student - and not the other way around. Steimann tries to overcome this problem by regarding a role's class as a generalization of the corresponding role filler's class. Hence, a class `Student` would be superclass of the class `Person`. While this construction would solve the problem formally, it that does not correspond to the common understanding. Steimann points out that all potential roles an object may play are part of its semantics. But even, if one agrees with this construction, regarding a delegator class as a generalization of the corresponding delegatee's class is not convincing, because one can hardly claim that, e. g., person adds further properties to student. Steimann himself speaks of a "paradoxical situation that, from the extensional point of view, roles are supertypes statically, while dynamically they are subtypes" (Steimann 2000b, p. 90). The third reason is both a conceptual and a technical one. There are cases, where only a delegator and its properties are required. For example, for some kinds of analysis, university administration is interested in students only regardless of the persons behind. In other words: it makes sense to abstract the delegatee away and regard the delegator as an object with an identity of its own. That recommends (not: implies) representing delegators as objects with an address outside the namespace of the delegatee object.

SR5 A delegator object must not be linked to more than one delegatee object at the same time. *Rationale*: There are cases, where it would increase flexibility, if there was a choice of delegatee objects at runtime. If, for example, a delegator object of the class `Interpreter` received the request to translate a text, it could dispatch the request either to a human or a machine, depending on the type of text. A further example would be scheduling: a customer requests a sales assistant (represented as delegator) and the corresponding object is linked dynamically to an available agent. Nevertheless, we decide against multiple delegatee objects, because it would require the specification of dispatch knowledge with delegator classes or the introduction of additional dispatchers. In both cases, the delegation of messages would get substantially more complicated. Instead, it seems preferable to perform the dispatch before sending a message to a delegator object. While this feature is not explicitly accounted for in Steimann (2000b), it seems that there is no concept of delegation or roles that would allow for multiple delegatees.

SR6 A delegator class may define properties (attributes, methods, associations) that already exist in a delegatee class it is associated with. *Rationale*: If the definition of properties with the same name does not happen by accident, it is a useful approach to override properties defined with a delegatee class and also to replace values specified in a delegatee object. Therefore, it should be allowed to use properties with the same name in associated delegator and delegatee classes (corresponds to #11 in Steimann (2000b)).

SR7 *(optional)* The convenient and safe definition of disjoint constraints on delegators should be supported. *Rationale*: As a default, delegators are overlapping, because this is an essential feature to overcome anomalies caused by specialization in object-oriented systems. For example, a person may fill simultaneously roles like student or programmer. However, there may be cases where it is important to express constraints that prohibit certain delegators from being taken simultaneously, e. g., professor and student. (see also SR8)

SR8 *(extension)* The specific semantics of variants should be supported. *Rationale*: If a delegator class is conceptualized as a variant, the delegation relationship implies the following constraint: The delegator classes need to

satisfy the disjoint constraint, since it makes no sense that an artifact serves as two variants of the same core artifact. In addition, it may be required that all variants of a core artifact are of the same kind. Many variant objects of the same kind may be linked to a core artifact object (see also SR2). Variants are not accounted for in the review in Steimann (2000b).

SR9 An object that serves as a delegatee in one delegation relationship may serve as a delegator in another delegation relationship. *Rationale*: Delegation hierarchies can be useful (see example in Fig. 5) and do not cause any specific problems (corresponds to #8 in Steimann (2000b)).

SR10 Cyclic delegations must not be permitted. In other words: by no means may a delegator object act as a (transitive) delegator of itself. *Rationale*: Apart from the fact that cycles would not make sense, they would jeopardize system integrity, because they may produce non-terminating message calls (not mentioned in Steimann (2000b)).

SR11 *(not essential)* Delegation should allow for virtual delegatee objects. *Rationale*: Delegation can be useful in cases where a delegatee objects does not represent any real domain object. In these cases, the delegatee object serves only the purpose to define values of certain properties for all its delegator objects. For example, the document types in an organization could be modelled as delegators of a virtual delegatee object of the class `Document` that defines values of properties such as footer or header (not mentioned in Steimann (2000b)).

2.3.2 Multi-level requirements

MR1 *(extension)* Delegation should also be available as an association between (meta-) classes on any classification level. *Rationale*: If delegation is used in a multi-level language, it should be modelled where it conceptually belongs. Hence, it should be possible to define delegation associations between classes on a meta-level (M2 or above). Note that this does not include associations between classes on different levels. As already pointed out, variants (see Fig. 7) are a good example. They are located on level M1 (or above), and not on M0. As a consequence, the corresponding delegation associates classes on a meta-level (M2 or above) (not accounted for in Steimann 2000b).

MR2 *(extension)* It should be possible to define delegation as a specific relationship between a class and its instances. *Rationale*: Classes may serve to represent values that are common to all their instances. Every instance should be able to access those values. These values could be accessed through methods offered by a class and the delegation of corresponding messages from the instances of that class (see example in Fig. 8). In those cases where the highest level of classification is M1, delegation to class offers only modest advantages over sending a message explicitly to a class – either by referring to the class like to any object or by using a special construct like `static` as in Java: an instance does not have to know about the methods in the class, which implies that changing the interface of a class would not require changing instance methods in that class. However, with a growing number of classification levels, delegation is getting more and more useful. If a certain value of a specific property is specified on a higher level of classification, it requires an intensive effort for objects on M0 to find the relevant level. An example would be a class `Carnivore` on M4 that defines for all instances that they feed on meat. An object that represents a particular animal, e. g., a fox, would not have to know the entire hierarchy, but simply dispatch a message like `feedsOnMeat` to its class, which would forward the message to its class, etc., until a class/object answers the message (not accounted for in Steimann (2000b)).

MR3 *(extension)* Delegation to class must not be modelled as delegation association. *Rationale*: Delegation to class is restricted to applying the

dynamic aspects of delegation to an "instance of" relationship. It is not a regular delegation relationship. This is for two reasons. First, there is no other conceptual relation than instance to class. Second, it would violate the constraints that an object must not be associated to more than one delegatee at the same time (SR2). This, however, would be possible, if an object served as a regular delegator of some delegatee object and at the same time as a "delegator" of its class. As a consequence, there should be other concepts in place that allow to decide whether forwarding a message to an object's class is an option at all (not accounted for in Steimann (2000b)).

2.3.3 Dynamic requirements

DR1 Message forwarding must be transparent, i. e., a message sent to a delegator object is processed equally by delegation as it would have been processed directly by the receiving object. Any change made to the delegatee interface will immediately be accessible to the delegator. *Rationale*: Even though this requirement creates a substantial implementation challenge, it is essential for the convenient and consistent use of delegation (not explicitly accounted for in Steimann 2000b, but addressed in #13). Transparent delegation is demanded for in Frank 2000 and is essential for prototype-based languages Lieberman 1986.

DR2 If a delegatee class and a superclass of a delegator class happen to have a method with the same signature and no other policy was defined, the inherited method should be selected. *Rationale*: Inherited properties can be seen as characteristic, invariant properties of a class. In that sense, they are not different from the methods defined in the class itself. Therefore, this rule corresponds directly to the basic rule that delegation takes place only, if a requested method is not included in the interface of an object. Note, however, that this rationale is not compulsory. Therefore, it should be possible to define other policies. This aspect is only indirectly accounted for in Steimann 2000b: there is no priority recommended because role types are considered as subtypes from a dynamic perspective.

DR3 If the state of a class does not include data that is representative for its instances, delegation to class is pointless und should be prevented. *Rationale*: If a class does not include any operation that could be applied to all of its instances, the attempt to delegate a message to this class will be a waste of resources.

DR4 If a delegator object receives a message, it cannot respond to, it should at first delegate it to its delegator object. If the delegator object cannot handle the message either, it should be dispatched to the delegator object's class. *Rationale*: There is no convincing justification for this policy in a particular case. It is merely based on the assumption that it will be the better choice on average because delegation to class is likely to work in fewer cases.

DR5 Delegatees assigned to delegators may change over time to reflect different behaviour of objects during their life cycle. *Rationale*: This requirement follows directly from SR.3.

Note that the list of dynamic requirements does not cover the decision between forwarding and delegation (see 2.2.1). From a conceptual perspective, we do not see a clear advantage of any of the two alternatives. Both qualify as an appropriate implementation of the conceptual idea of delegation. Both have specific advantages and shortcomings. Therefore, it is important that users of a modeling tool are informed about possible pitfalls of the specific implementation.

2.3.4 Tooling requirements

TR1 If an object is manually instantiated from a delegator class during modelling, the user should be supported with selecting or creating a suitable delegatee object. *Rationale*: Different from specialization, delegation relationships

have to be established during runtime. This will often happen on a user's request. Without support, the instantiation of delegator objects would not only stress users, it would also jeopardize system integrity.

TR2 There should be a mechanism to retrieve the list of all methods available with a delegator class. In addition, it should be possible to retrieve those methods only that become available through delegation. *Rationale*: This is relevant to know for understanding the behavior of objects instantiated from a delegator class. Being able to distinguish between different origins of available methods is also vital for effective debugging support in systems that use delegation. With respect to delegation to class, this requirement implies that an object should be able to provide a list of those methods of its class that give access to values representative for all instances.

Even though we have used various forms of delegation for almost two decades, we are not sure whether further requirements will emerge in the future. Therefore, we suggest a further "meta" requirement: an implementation of delegation should allow for adding further constraints.

2.4 An Extended Concept of Delegation

To specify static aspects of delegation that satisfy the above requirements, we will use a simplified meta model of an object-oriented modelling and programming language. XCore (Clark et al. 2008a), see Fig. 11, features a recursive architecture and enables multiple levels of classification. Therefore, it is also suited to cover the use of delegation in multi-level environments. At the same time, it can be applied to languages that are restricted to classes on M1. The central class `Class` is, by default, located on M2. However, it may also represent classes on any level above M3. Therefore, its level is contingent. To apply the meta-model to languages that do not support multiple levels, `Class` can be assumed to be located on M2 only. Delegation is modelled as a kind of association (attribute `type` of `Association`). The conceptualization as role / role filler or variant / core artifact is defined through the attribute `role` of `End`.

The attribute `isRepresentative` outlines how to cope with delegation to class. If `isRepresentative` is true, messages sent to an instance of the corresponding class, can be delegated to that class.

3 Application of Delegation

The above use cases demonstrate that delegation is suited to overcome problems with inheritance and specialization caused by the specific notion of class in object-oriented systems. However, its successful use requires appropriate design decisions. In particular, there is need for criteria that support the identification of associations that qualify as delegation. That includes the distinction of delegation and specialization. There is only little support for the adequate use of delegation in the literature. Gamma et al. (1994) characterize delegation as powerful but avoid providing any criterion for choosing between delegation and inheritance: "Delegation is a good design choice only when it simplifies more than it complicates. It isn't easy to give rules that tell you exactly when to use delegation, because how effective it will be depends on the context and how much experience you have with it." (Gamma et al. 1994, p. 32). While the conceptualization of delegation associations with roles/role fillers, and variant/core artifacts respectively serves as a useful orientation, it is not sufficient. To provide more effective support, we will first take a closer look at the meaning of role and variant. Subsequently, we present pragmatic guidelines to support design decisions related to delegation.

3.1 Roles and Variants

There are numerous definitions of the term "variant". In general, a variant represents a modification of some kind of core artifact, which itself may be a variant. With respect to its use within a delegation association, we propose the following definition. An object representing a variant

depends on another object that represents a core artifact. The variant object has the same properties and property values as the object that represents the core artifact. It may override values defined for the core artifact object. The definition of the role is more demanding. Guarino et al. (1994) aims at developing an ontologically founded concept of role that allows distinguishing roles from other objects. He regards roles as concepts. With respect to the terminology we used so far, "role" should be replaced by "role type". For the definition of this concept, Guarino et al.. introduce two different kinds of concepts. A concept α is *founded* (on another concept β), if every instance of α must be related to an instance of β, without being a part of the instance of β (Guarino et al. 1994, p. 5). A concept α is *semantically rigid*, if its instances have to be of α during their entire lifetime, otherwise they would lose their identity. *Person* would be an example of a semantically rigid concept, while *Child* would not. "A concept α is called a role if it is founded but not semantically rigid ..." (Guarino et al. 1994, p. 6). This conception of role is definitely useful to identify role candidates. However, its application is not trivial, and in part, misleading. The definition of semantically rigid objects is in so far misleading as in object-oriented systems an object that represents a role has a different identity than the corresponding role filler object.

To develop more pragmatic criteria to support the identification of roles, we will look at the relationship between delegation and specialization or inheritance respectively. This is required anyhow, because various authors use this relationship to describe roles or delegation, and the corresponding proposals are not consistent. Lieberman (1986) regards delegation to be more powerful than inheritance, while Stein comes to the conclusion that "delegation is inheritance" (Stein 1987). Nevertheless, she suggests choosing between the two "depending on the needs of the application" (Stein 1987, p. 144). However, she does not provide any hint on when delegation should be preferred over inheritance. The logician Sowa does not speak of delegation. But he proposes a definition of roles that is based on specialization: "role types [...] are subtypes of natural types", "TEACHER is a subtype of PERSON in the role of teaching" (Sowa 1988, p. 120). Such a conceptualization is confusing for various reasons. It seems to be in conflict with the counter-intuitive effects of specialization we discussed in Sect. 2.1, and it leaves us with the question, of whether there is a formal difference between delegation and specialization. Furthermore, it is in obvious contrast to Steimann's view on role types as a generalization of entity types. Nevertheless, we shall see that regarding roles as subtypes leads to useful criteria to conceptualize roles – and to discriminate delegation against specialization.

To analyze the relationship between delegation and specialization we do not use the specific notion of specialization in object-oriented systems. Instead, we refer to the concept of specialization in SQL. This is for two reasons. First, it corresponds more clearly to the notion of specialization in logic and natural language than that in object-oriented systems. Second, its semantics can be systematically variegated. Therefore, it enables a more differentiated comparison of delegation and specialization. In SQL, the semantics of specialization can be variegated by "completeness" and "disjointness" constraints. A completeness constraint serves to define whether specialization is total or partial. It is total if the generalized entity type must not have any instances of its own. In the case of partial specialization, the generalized entity type may have instances of its own. A disjointness constraint allows us to define whether specialized entity types may overlap or not. Fig. 9 illustrates partial specialization where the specialized entity types are overlapping. It is clearly different from specialization in object-oriented systems, because an entity may belong to more than one type simultaneously. In particular, every entity of a subtype is an entity of its supertype, and an entity of subtype *a* may be an entity of subtype *b*, too.

The four different kinds of specialization in SQL provide a solid and easy orientation for supporting

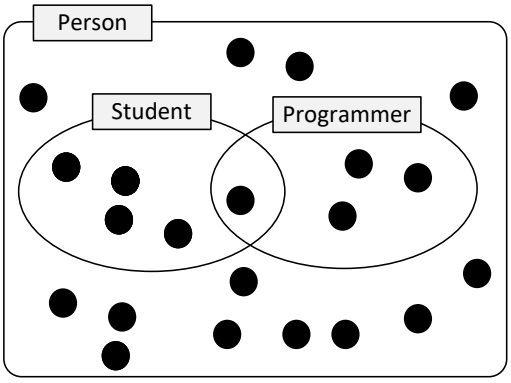

Figure 9: Partial specialization, overlapping

the decision between delegation and specialization in object-oriented systems. Tab. 2 depicts the default recommendations for each of the four different kinds of specialization. Whenever a partial specialization is identified, the subclass qualifies as a role class or, in other words, delegation is preferable over specialization. It seems to reflect the conceptualization of Guarino et al. (1994). However, it is easier and more clearly to identify.

Total specialization corresponds to the specialization of abstract classes in object-oriented systems. Therefore, it is not suited for delegation, because delegation depends on a delegatee object, that is, an instance of the superclass would be mandatory. However, in the case of overlapping specialized classes, specialization in object-oriented systems would not be satisfactory, because that would imply that there were objects that are instances of all overlapping specialized classes – which is not possible in object-oriented languages. Multiple inheritance would not be satisfactory either. If, for example, one would create the subclass `StudentProgrammer` from the classes `Student` and `Programmer`, the instances of that class would not be instances of the two superclasses, which is likely to produce redundancy and compromise integrity (see introduction).

We can summarize that delegation shares commonalities with specialization. However, at the same time it is different, in contradiction to the conclusion of Stein (1987). Also, different from the claim made by Lieberman (1986), delegation is not more expressive than specialization. This assertion can easily be proven informally: as we have seen, the case of an abstract superclass (or total specialization respectively) cannot be equivalently represented with delegation.

3.2 Guidelines

The following guidelines are based on the above conceptualization of roles and variants. The first group of guidelines relates to an early phase of domain analysis. These guidelines raise awareness for delegation and help with the identification of possible candidates for delegation.

G1 Do not get confused by the ambiguity of *is a*. Ask yourself whether a relationship between two concepts could also be called *represents* or *acts as* respectively. If this is the case, you have found a delegation candidate.

G2 Delegation is closely related to the common sense concept of a role. Notions such as *task*, *job*, *serves as*, *works as*, *etc*. may indicate a relationship between a role filler and a role. Therefore you should look for corresponding terms within available descriptions of a domain.

G3 Whenever you encounter the existence of different views on an object or different contexts an object may be assigned to, it is a good idea to check whether these views or contexts can be related to the roles or responsibilities of the object in a natural way. In this case, delegation might be a useful option.

G4 Some real-world entities are likely candidates for becoming role filler objects: persons, organizations, and versatile machines. Assigning the objects of a preliminary object model to such categories may help with identifying delegation associations.

G5 If a product or some other artifact represents a variation of another, more general artifact, it might represent a variant. In that case, it should be checked whether it falls under the specific concept of variant presented above.

Kind of specialization in SQL	Recommended use
partial, overlapping	delegation
partial, disjoint	delegation
total, overlapping	?
total, disjoint	specialization

Table 2: Different kinds of specialization in SQL

G6 If specialization seems to be a natural choice, go through the criteria in Tab. 2 and check if it is applicable. In cases where partial specialization appears useful at first sight, delegation is likely to be a better choice.

G7 If all instances of a class are characterized by the same value for certain properties, it might be a good idea to store this value with the class and apply delegation to the class to enable transparent access to this value.

The second group of guidelines aims at the design phase. They serve to check the preliminary results of the domain analysis and support final design decisions.

G8 In those cases, where the above criteria recommend the use of delegation, check whether the prospective delegator objects may change during the lifetime of the corresponding delegatee object. If that is definitely not the case and the implementation language you use does not provide support for delegation, specialization is a pragmatic, but valid option.

G9 If a pragmatic use of specialization is not an option and the implementation language does not support delegation, look for appropriate design patterns that enable to mimic delegation. Never use specialization only for the reason that delegation is not available, because you would risk unpleasant effects (redundancy, threat to integrity).

G10 If an object qualifies as a variant of some core artifact, check whether variant and core artifact should be modelled as classes. In that case, delegation should be ideally specified on a meta-class level (see fig 7). If that is not an option, classes could be represented as objects on M0, which would, however, imply introducing an artificial instantiation relationship between these objects and other objects that represent their "instances".

Delegation and specialization can be combined. This is the case if a delegator class qualifies as a possible subclass or a delegatee class qualifies as a possible superclass. In the example in Fig. 10, `Professor` is specialized from `Employee`. This could make sense, if in a corresponding system, every represented professor must be an employee at the same time, and can never become an employee of a different kind or just an employee (G8). Even though delegation would be a valid option, too, specialization has the advantage of enforcing the constraint that every professor has to be an employee already at compile time.

4 Language Architecture for Delegation

The concept of delegation can be realized in a number of different ways all of which satisfy some or all of the requirements discussed in Sect. 2.3. Implementations are likely to differ with respect to object arrangements, class organization, integration with inheritance, and message-passing mechanisms. Our aim is to provide a language framework for delegation within which the different approaches can be constructed, analyzed and compared. We have used the XModeler platform (Clark et al. 2008a; Clark and Willans 2012) as a basis for delegation analysis because it

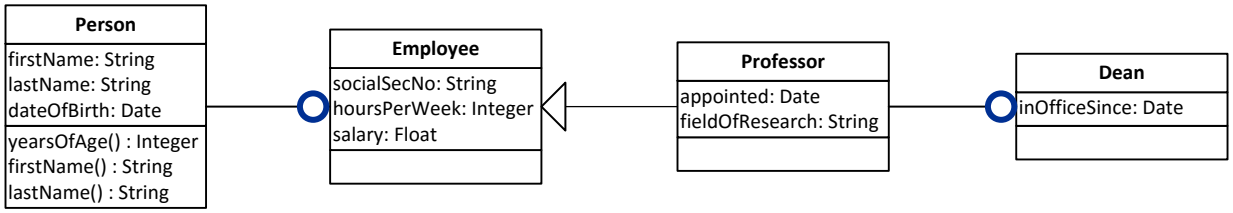

Figure 10: Combined use of delegation and specialization

has been designed as an extensible language engineering environment that offers visual modeling capabilities together with a comprehensive (meta-)programming language. In particular, the key structural and dynamic features of standard object-oriented class-based languages can be modified and extended within XMODELER.

Our delegation architecture and the examples of its use in this article have been implemented in the XOCL language in XModeler. XOCL combines features from the Object Constraint Language (OCL), with an action language and features from functional programming. The semantics of XOCL is extensible because it is based on a meta-object protocol (MOP), and the syntax of XOCL is extensible through the use of syntax classes. Both the MOP and syntax classes have been used in our implementation of the proposed delegation architecture; therefore, whilst it is not essential to understanding our contribution, we include the code as an aid for readers who are interested in the implementation details.

This section describes the essential language features of XMODELER, to create a foundation for the later explanation of different implementation options for delegation. Sect. 4.1 defines the kernel of XMODELER in terms of a collection of self-describing classes called XCore and in terms of its extensible operational semantics. XMODELER provides a programming language called XOCL that can be extended with new language features as described in Sect. 4.2. The extensible features of XMODELER will be used in the following section to build two different implementations of delegation.

4.1 Language Architecture and XMODELER

XMODELER is a meta-modelling environment that supports language engineering and executable modelling. It runs on a virtual machine (VM) that supports simple data types and associated operations. A kernel model called XCore and a kernel language called XOCL are defined in terms of the services offered by the VM. All other aspects of XMODELER are defined in terms of XCore and XOCL.

XMODELER runs on a small VM that is implemented in Java; the VM manipulates data structures that are defined in Fig. 20 in the appendix. The XOCL language is defined in Fig. 21 in the appendix and compiles (using a compiler written in XOCL) to VM instructions that run in terms of the data structures defined in Fig. 20. A small collection of kernel operations are made available in XOCL to manipulate the VM data which is otherwise hidden from the user. Functions contain machine instructions, globals (closed in values), dynamics (imported name-spaces), the self-object, arbitrary properties, a type signature, and super-functions.

The semantics of XOCL is made extensible by defining the operational rules in terms of messages sent to a set of pre-defined classes called XCore. The operational rules are defined by the interpreter shown in Fig. 23 in the appendix and key XCore classes are shown in Fig. 11. The figure uses standard notation for classes expressed as boxes with each a compartment for the name of the class, its properties, and its operations. In XCore there is no semantic difference between properties and directed edges between classes, except that

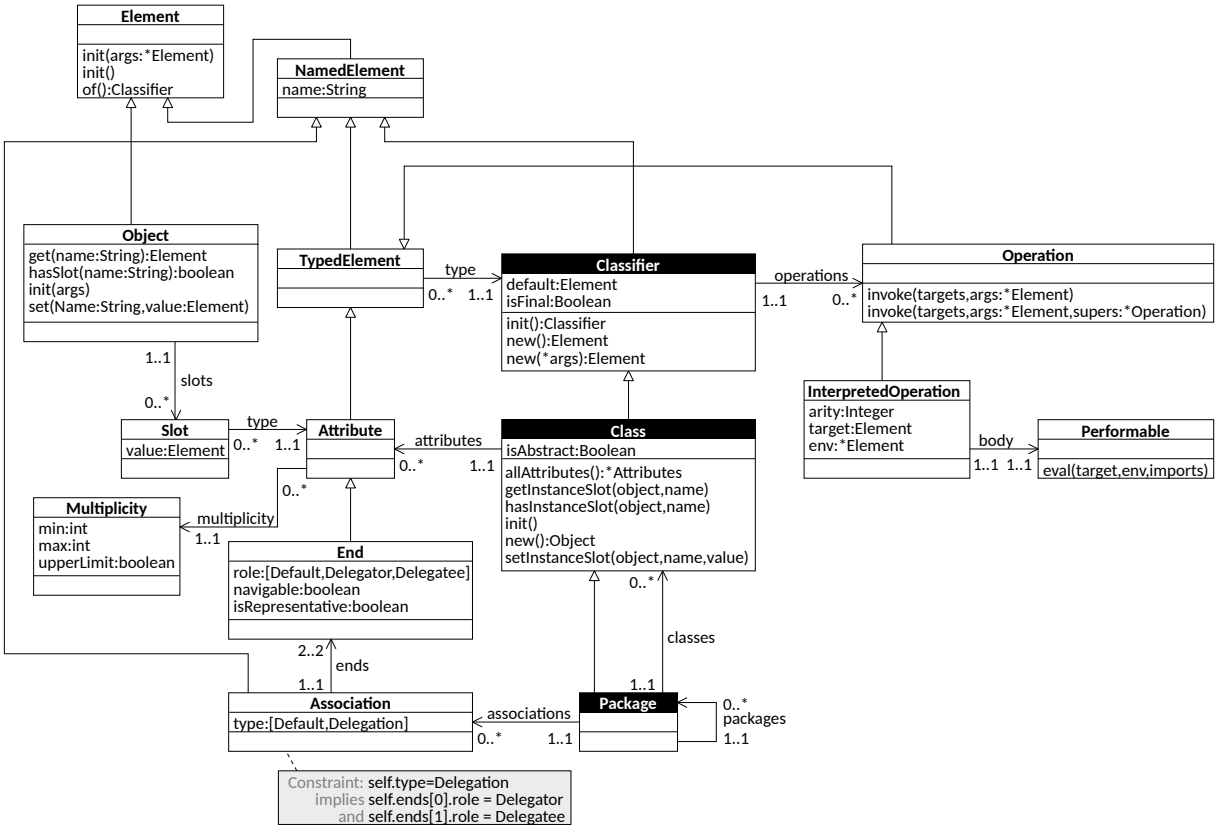

Figure 11: XCore classes and selected properties

the latter can be decorated with multiplicity at the source and the end. The type of properties and operation arguments may be omitted when it is the default type `Element`. A prefix * on an operation argument means that the supplied value is expected to be a collection.

XCore distinguishes between *elements* and *objects*, and correspondingly *classifiers* and *classes*. An element (such as the integer `1`) does not have a state whereas an object (such as the point `(10,20)`) has state represented using slots.

An XCore meta-class inherits from the distinguished class `Class` and therefore inherits the operation `new` that allows its instances to create new objects. Meta-classes are indicated with a highlighted name box in the diagram, showing the name of the meta-class printed in white on a black background.

Consider a sequence of integers, its classifier is `Seq(Integer)`. The classifier of `Seq(Integer)` is `Seq`, whose classifier is `Classifier` whose class is `Class`. The class of `Class` is itself.

Operational extensibility is achieved in Fig. 23 through the definition of `App`, `Send`, `Get`, and `Set`, each of which rely on sending messages whose basic definitions are associated with the classes in Fig. 11.

XCore classes provide a meta-model for an object-oriented language. In Fig. 11, structural features of classes are represented by attributes, unidirectional associations and bidirectional associations. Behavioral features are represented by operations that can be compiled or interpreted. All data in XCore must be represented using the VM types defined in Fig. 20 in the appendix, including classes and meta-classes, therefore the

definitions shown in Fig. 11 must be bootstrapped using kernel-level operations. A representative part of the bootstrap is shown in Fig. 22 in the appendix. Once bootstrapped, XCore serves as a dynamic type system for XOCL. Classes are created by sending `Class` a `new` message and are populated by creating instances of `Attribute` and `Operation`.

XCore provides a meta-object protocol (MOP) that is used to make instance representation and behaviour extensible. Figures 12, 13 and 14 show the XCore operations that are required to support the evaluation mechanism for XOCL. In each case a definition is of the form `C::n(args) exp` where `C` is the class that owns the operation named `n`. For example the definition of `Object::get` checks whether the meta-class of the receiving object is `Class` in which case the kernel operation `Kernel_getSlotValue` is used to directly access the VM object representation; otherwise, the class of the object is sent a message `getInstanceSlot` allowing different types of class to define different access strategies. Sending messages is defined by `Element::send` which either finds an operation via `allOperations` or sends a `noOperationFound` message which can be redefined for different types of objects.

4.2 Syntax Classes

XCore implements a textual language architecture which associates *syntax classes* with grammars. A syntax class named `C` defines a new language construct of the form `@C ...`, where the text denoted by the ellipses is processed by `C.grammar`. Fig. 15 shows the basic features of syntax classes: a grammar is associated with rules that parse text and synthesize performable elements via actions. The XCore parser reads program text and returns a performable element that is subsequently compiled. When the parser encounters `@C ...` it hands control over to `C.grammar` and expects a performable element that is inserted into the parser's output. A grammar may inherit from multiple parent grammars in which case they include the parent rules. An XCore class of type `Sugar` extends the abstract syntax of XCore and

```
Element::of()   = Kernel_of(self)
Element::init() = self

Element::send(name,args) =
  // If the meta-class of the target
    element is
  // Classifier, then we can use the
    default mechanism
  // to lookup and invoke an operation...
  if of().of() = Classifier
  then
    let ops = of().allOperations().select(
    fun(o)
          o.matches(name,args.size()))
    in if ops = []
       then self.noOperationFound(message,
    args)
       else ops.head.invoke(self,args,ops)
  else
    // Otherwise use the definition of
    sendInstance
    // provided by the class of the target
    ...
    of().sendInstance(self,message,args)

Element::noOperationFound(message,args) =
  // The default handler for undefined
    messages.
  // This can be redefined by any class...
  throw Exceptions::NotFound(self,message,
    args)

Object::get(name) =
  // If the meta-class of the target of
    the field
  // reference is Class, then we can use
    the default
  // mechanism...
  if of().of() = Class
  then Kernel_getSlotValue(self,name)
  else
    // Otherwise use the definition of
    getInstanceSlot
    // provided by the class of the target
    ...
    of().getInstanceSlot(self,name)

Object::hasSlot(name) =
  if of().of() = Class
  then Kernel_hasSlot(self,name)
  else of().hasInstanceSlot(self,name)

Object::set(name,value) =
  if of().of() = Class
  then Kernel_setSlotValue(self,name,value
    )
  else of().setInstanceSlot(self,name,
    value)
```

Figure 12: XCore MOP (Part 1 of 3)

```
Object::init(args) =
  // The constructors of a class are used
    to
  // initialise a new instance. Choose a
    constructor
  // based on the number of supplied
    initialisation args...
  let C = of().allConstructors()
      cnstrs = C.select(fun(c) c.names.
    size()=args.size())
  in if cnstrs = [] then self
     else cnstrs.head.invoke(self,args);
     self.init()

// The following are examples of sequence
    operations
// and are provided so that the
    definitions are self-
// contained...

Seq(Element)::select(predicate) =
  if self = [] then self
  else if predicate(head())
       then Seq{head | tail.select(
    predicate)}
       else tail.select(predicate)

Seq(Element)::size() =
  if self = [] then 0 else 1 + tail.size()

Operation::invoke(target,args) =
  // Operation invocation is a kernel-
    level
  // activity...
  Kernel_invoke(self,target,args,supers)

// The following two operations show how
    class-level
// properties are inherited...
Classifier::allOperations() = operations +
  rmdups([o | p <- parents,o <- p.
    allOperations()])

Class::allConstructors() = constructors +
  rmdups([c | p <- parents,c <- p.
    allConstructors()])

Class::allAttributes() =
  attributes +
  rmdups([a | p <- parents,a <- p.
    allAttributes()])
```

Figure 13: XCore MOP (Part 2 of 3)

```
// ************ Start of the Default MOP
    *************

Classifier::sendInstance(element,name,args
    ) =
  // This is the default mechanism that is
    used to deliver
  // messages. It can be redefined in meta
    -classes in
  // order to define new meta-object
    protocols...
  let ops = element.of().allOperations()
      op = null
  in while ops <> [] and op = null do
       op := ops.head;o
       if not(op.matches(name,args.size())
    )
       then op := null;
       ops := ops.tail
     end;
     if op <> null
     then op.setSupers(ops);
          op.invoke(element,args)
     else element.noOperationFound(name,
    args)

Class::getInstanceSlot(object,name) =
  Kernel_getSlotValue(object,name)

Class::hasInstanceSlot(object,name) =
  Kernel_hasSlot(object,name)

Class::setInstanceSlot(object,name,value)
    =
  Kernel_setSlotValue(object,name,value)

Class::new() =
  let A = allAttributes()
      mkSlot(a) =
        let n = a.name
            v = a.type.default
        in Kernel_mkSlot(n,v)
      slots = [ mkSlot(a) | a <- A ]
  in Kernel_mkObj(self,slots).init()

Class::new(args) = new().init(args)

// ************ End of the Default MOP
    *************

// Given all the definitions above,
    objects can be
// created, accessed, updated and sent
    messages. Since
// all the operations are attached to meta
    -classes,
// they can be redefined to define new
    MOPs.
```

Figure 14: XCore MOP (Part 3 of 3)

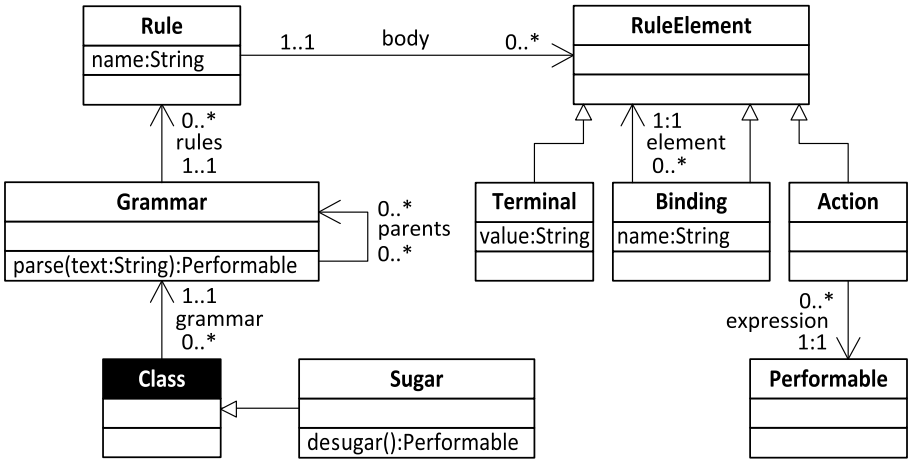

Figure 15: Syntax classes

must supply a `desugar` operation that returns a performable element that is known to the compiler. This mechanism makes XOCL easily extensible, for example if we want to add a new language construct that supports state machines, we can define a declarative model for state machines that is created via a syntax-class `Machine` whose grammar parses a new language construct, for example `@Machine S1 -t1→ S2` **end**. The grammar synthesizes an instance of the state-machine model that is translated via its `desugar()` operation into basic XOCL, for example an operation that is supplied with a transition name and returns the next state as defined by the machine so that:

```
m :=
  @Machine
    S1: t1 → S2;
    S2: t2 → S3
  end;
m("t1") => S2
m("t2") => S3
```

5 Implementation of Delegation

The previous section has described the extensible object-oriented language framework XMODELER. This section examines how delegation can be implemented in XMODELER using two different approaches. The first implementation option discussed in Sect. 5.1 delegates messages by extending the error handling mechanism of XCore. The second approach in Sect. 5.2 implements delegation by redefining the XCore message handling with an extended meta-object protocol for delegation. Both approaches have their benefits and drawbacks and can be used as the basis of further study.

5.1 Implementing Delegation by Handling Message Failures

Given a standard object-oriented message-handling mechanism, we may choose to provide delegation by waiting until something goes wrong with regular message dispatching and then using the language's exception-handling mechanisms to implement the semantics of delegation. This approach turns out to be easy to realize in cases where the appropriate level of operational control is exposed by the underlying programming language. XOCL is one of the languages that allow intercepting the message passing process by means of exception handlers. Exception handlers consist of code that gets executed in response to abnormal conditions that occur during the execution of the regular method code.

Fig. 16 shows a schematic sketch of how a delegation mechanism can be realized based on exception handling capabilities. The situation distinguishes between one side of the program

execution which is under control of a programming language interpreter or compiler, and one side which is controlled by programs written in the respective programming language. These are labeled "Execution" and "Language" in the figure, respectively. Any machine executable implementation of a programming language generally can be decomposed into these two components, no matter whether the language is interpreted at runtime, or a compiler generates the execution behavior at build time. When during the execution of a program an operation is invoked from the language side, a dispatching mechanism embedded in the execution side takes over and passes the control focus to that invoked operation. Some languages provide a fixed implementation of the dispatcher mechanism that always requires a unique object reference and an unambiguously referenced operation to be invoked. More flexibility is provided by languages that allow to intercept the behavior of the dispatching mechanism.

The dispatcher raises exceptions in cases when an unexpected situation occurs during operation dispatching, e. g., when an operation is not available on the given object reference. Fig. 16 exemplifies this situation with a `noOperationFound` exception, which is raised internally on the language execution side in case an operation cannot be found, and causes program control to be passed over to the language side for handling this exceptional case. By default, a typical object-oriented language treats the case of an unavailable operation as a severe fault and reacts with an error message and program termination. This behavior of "Default Exception Handlers", as labeled in the figure, is part of the language execution side. It can be overwritten by user-defined exception handlers that behave differently and react to the exception in a controlled way. The delegation semantics can be implemented in this way by passing the control flow to an operation of an object that is different from the originally targeted one.

If an exception of type `noOperationFound` is incorporated in the set of exception types explicated by the execution mechanism, then raising such an exception propagates the information about a missing operation into the realm of user-controllable program execution. An implementation of delegation can then determine whether the object executing the operation is linked to an object that serves as a delegatee, and in that case forward the invokation to the delegatee object. The following code excerpt exemplifies this mode of operation:

```
context Element
 @Operation
   noOperationFound(message:String,args:
    Seq(Element))
   // If the receiver doesn't understand
    the message,
   // it propagates it to its delegatee.
   if delegatee = null
   then throw Exceptions::NoOperation(self
    ,message,args)
   else delegatee.send(message,args)
   end
 end
```

Intercepting the message dispatching context at this late point in time, after a missing operation has already been detected, comes with the advantage of not requiring the language execution mechanism to take care of how the delegation behavior is specified. Instead, the execution core only signals the lack of an operation by conveying this to the level of program-controllable behaviour via an exception. While the language defines some canonical default behaviour for cases in which exceptions are raised, simply by replacing the handler code for these types of exceptions, all actions that characterize the semantics of delegation can now be defined using regular programming language constructs of XOCL, and can be provided by a behavior extension specified through an exception handler.

A potential disadvantage potentially associated with this approach is the lack of contextual information about the state of the execution engine by the time the exception, i. e., the lack of an operation on an object was detected. Such information, e. g., the instance reference to the calling object that initiated the method invocation, can especially be valuable when propagating the message call through a hierarchy of delegatee objects. Also for

debugging purposes, it would be desirable to have access to this information. However, this is not an inherent consequence of this implementation. It can be avoided by implementing delegation as a bidirectional association.

Another potential disadvantage of the approach is that the default message-handling mechanism provided by the language must be completed before the delegation mechanism can take over. This means that there is no way of circumventing inheritance: if inheritance finds an operation, then delegation is overridden. That is, however, not a problem as long as messages should at first be dispatched to the superclass anyway.

Keywords like `self` within operations that are executed by a delegatee object upon a message dispatched from a delegator object refer to the delegatee object, not to the delegator object. While this makes a difference in extremely few cases only, it is important to know to avoid inconsistent behaviour (cf. the discussion of delegation versus forwarding in 2.2.1).

A similar approach can be used to allow delegation to access data slots of a delegatee. Like message handling, a `noSlotFound` exception is raised when a slot is not available in a target object, and the associated operation can be redefined. Like intercepting the message handling, the key advantage is that this approach is simple, and the disadvantage is that the mechanism must wait until all standard slot access is complete, meaning that it cannot override inheritance and may not have access to the appropriate contextual information.

5.2 Implementing Delegation by a Meta-Object Protocol

A key disadvantage of the approach described in the previous Sect. is that delegation must wait until all existing message passing and slot access mechanisms have been completed. This prevents delegation from being able to override inheritance, for example. Any delegation mechanism that requires more fine-grained control over language execution must find a way of replacing existing operational features.

Most standard languages do not support such replacement or overriding since their operational features are fixed in terms of an interpreter or a compiler. XOCL provides access to its key operational features in terms of a *meta-object protocol* (MOP) (Kiczales et al. 1991) whereby classes and associated operations that implement basic features of the language are available to the developer and can be replaced or extended systematically. Where a MOP is defined for a particular purpose, such as database access, it is often referred to as a domain-specific MOP, e. g., a *database*-MOP.

Fig. 17 shows a schematic sketch of how a delegation mechanism can be realized based on a MOP. The user program contains an operation call, a slot access, and a slot-update: an *object protocol* (OP) with respect to a *receiver* object. Without the use of a MOP, an underlying OO language engine will implement an OP in a fixed way that cannot be modified or extended by the user. An MOP replaces a fixed OP with three dispatchers each of which uses the type of the receiver as the target of normal operation calls: `sendInstance`, `getInstanceSlot`, `setInstanceSlot`. Since the receiver of these operations is the type of the original receiver, the MOP is defined at the meta-level. A standard MOP, defined by `Class`, implements the original OP. The user is now free to implement any number of new MOPs, for example in class `Role` to implement delegation.

Compare the MOP-based approach shown in Fig. 17 with that shown in Fig. 16 that uses message failure to perform delegation. A key advantage of the former is that it handles message delivery and slot access using an interface that is provided to extend the language semantics, whereas the latter uses an interface that is intended to handle errors. In that sense, the ability to override slot access in order to provide delegation is a result of the particular implementation of error handling, which cannot be relied upon. It would be reasonable to change the universal error handling mechanism to interrupt the current flow of control, in which case the control would not return to the original dispatch.

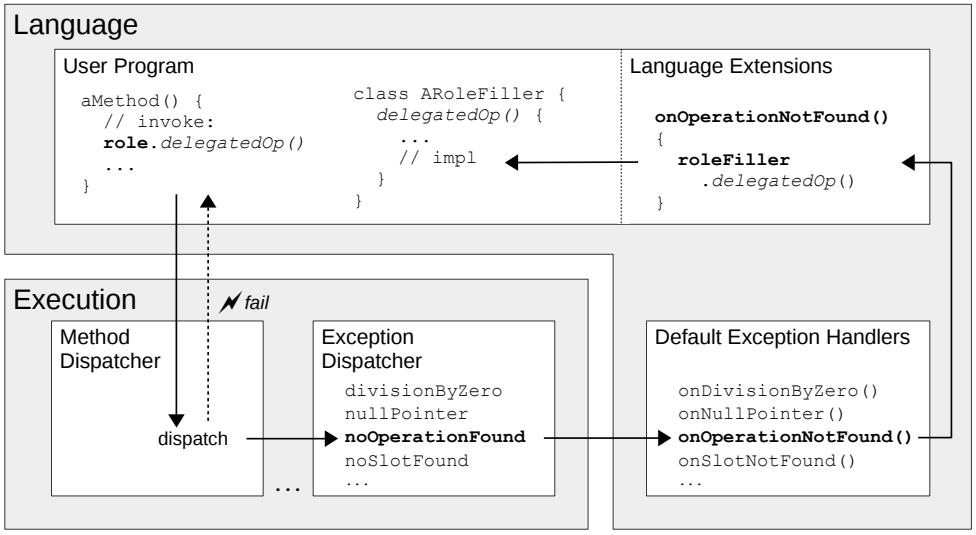

Figure 16: Delegation handled via message failure

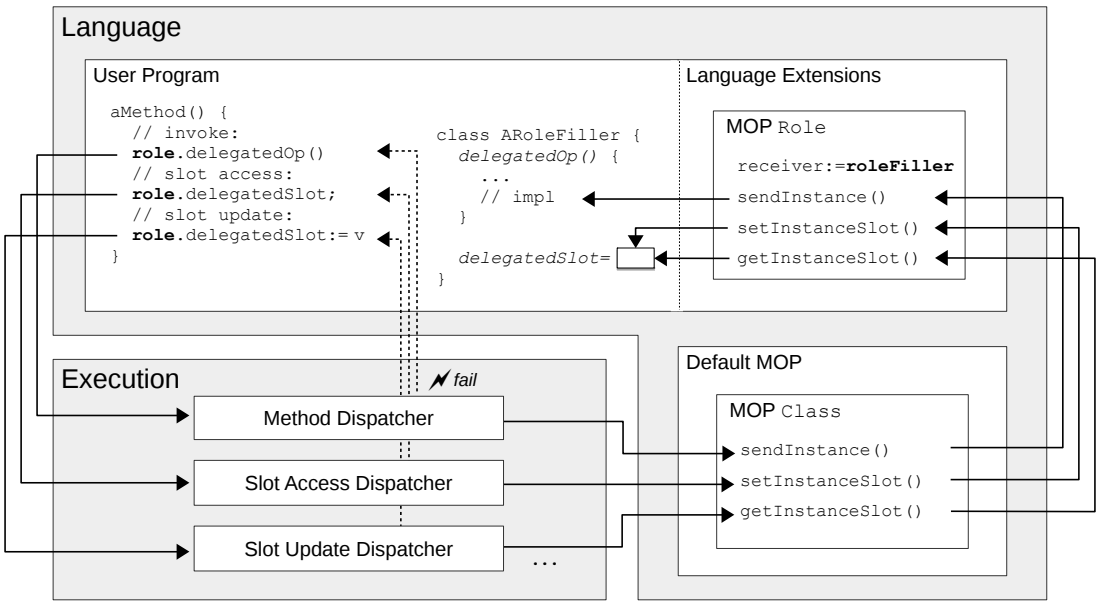

Figure 17: Delegation handled via meta object protocol

A MOP does not rely on such implementation concerns since it is guaranteed to return to the original dispatch request (unless the user implements a handler to do otherwise). Furthermore, a logical definition of delegation would naturally be defined in terms of message passing and slot access semantics, not in terms of error handling. A MOP is a language mechanism that allows the key semantic features of a language to be extended and therefore is arguably the right choice to define delegation. This is demonstrated by the implementation described in the following sub-Sec.s where we extend the notion of delegation with the idea of *redelegation* along a delegation-chain in much the same way as *super* is used in JAVA to continue along an inheritance-chain.

A MOP provides a mechanism to specify the operational semantics of the `.` operator that accesses and updates slots. This is achieved by defining an operation in the meta-class of an object (the class of its class) that defines how the `.` operator should be handled. Operationally, this satisfies the requirements of delegation which needs to redirect slot requests from one object to another.

The particular delegation MOP used in this article provides unrestricted slot access, however, access could be restricted by extending the definition of attributes to include access protocols that are satisfied by the implementations of `setInstanceSlot` and `getInstanceSlot`. Such access protocols would be different from those that affect inheritance and thereby limit access to slots in subclasses.

As described in Sec. 4.1, XMODELER provides an MOP as defined by the meta-class `Class`. The following sub-Sec.s define a *delegation*-MOP using the features of XMODELER, starting with a motivating example.

5.2.1 Delegation chain example

Consider the delegation-based model in Fig. 5. A person has a name and a pay-rate. The cost of a person, based on the notion of a minimum wage, is their pay-rate multiplied by the duration of time over which they work. At some point in their life, a person may become a lecturer and

```
context Root
  @Class Person
    @Attribute name : String
                  end
    @Attribute payRate : Integer
                  end
    @Constructor(name,payRate) !
                  end
    @Operation getName():String name
                  end
    @Operation setName(n:String) name := n
                  end
    @Operation getPayRate():Integer
    payRate             end
    @Operation setPayRate(n:Integer)
    payRate := n        end
    @Operation cost(dur:Integer)
    getPayRate() * dur      end
  end

  @Class Lecturer metaclass Delegator
    @Attribute subject : String
                  end
    @Attribute payRate : Integer
                  end
    @Slot Delegator::delegatesTo = Person
                  end
    @Constructor(delegatee,subject,payRate
    ) !           end
    @Operation getSubject() subject
                  end
    @Operation setSubject(n:String)
    subject := n        end
    @DelegatorOp getPayRate() payRate +
    redelegate()/10 end
  end

  @Class Dean metaclass Delegator
    @Attribute payRate : Integer
                  end
    @Slot Delegator::delegatesTo =
    Lecturer            end
    @Constructor(delegatee,payRate) !
                  end
    @DelegatorOp getPayRate() payRate +
    redelegate()/10 end
  end
```

Figure 18: Motivating example (XOCL)

their pay-rate changes during the time that they hold an academic post. In addition, a lecturer may become a faculty dean. This may occur for a fixed duration of, say, 3 years, during which time their pay-rate is further increased.

The XOCL implementation in Fig. 18 shows that a person is defined in a standard way and produces a class with attributes, accessors, updaters and a constructor. The operations defined for `Person` all reference slots and operations with respect to `self`, which under conventional circumstances will be an instance of `Person` or one of its subclasses.

The role `Lecturer` defines two attributes: `subject` and `payRate`. The delegatee for any `Lecturer` must be an instance of `Person` such that the slot named `payRate` in the `Lecturer` shadows the slot named `payRate` in the `Person`. The operation named `getPayRate` is defined in `Lecturer` to return the lecturer's pay-rate augmented with one-tenth of the standard person-rate. The operation is defined using the special syntax `DelegatorOp` which means that there is a special value called `redelegate` that can be used to re-delegate the message.

The role `Dean` uses a `Lecturer` as a delegatee and uses a local definition of `payRate`. A dean may therefore have a pay-rate of `3`, and a delegatee which is a lecturer with a pay-rate of `2`, and a delegatee which is a person with a pay-rate of `1`.

Sending a message `cost(10)` to a dean proceeds as follows. The role `Dean` does not define or inherit an operation named `cost` so the message is delegated to the corresponding instance of `Lecturer`. Since `Lecturer` does not implement the operation, the message is delegated to the instance of `Person` where the operation is found. The body of `Person::cost` multiplies the duration (`=10`) by the result of sending `self` a message `getPayRate`. At this point, although we are in the class `Person`, the value of `self` is an instance of `Dean` which was the target of the original message. `Dean` defines `getPayRate` as a role-operation that adds the local value of `payRate` (`=3`) to the result of re-delegating the message to the lecturer. Since `Lecturer` defines an operation named `getPayRate`, it handles the message, and since the message was re-delegated, the value of `self` in `Lecturer::getPayRate` is the corresponding instance of `Lecturer` leading to the local value of `payRate` (`=2`) being added to the result of further re-delegating the message. Finally, the message is handled in `Person` where the value of `1` is returned. The complete calculation results in a value of `32.1`.

5.2.2 Redelegation

Inheritance and delegation are similar with respect to message passing in that both cause operations to be combined in two different dimensions. Inheritance can be viewed as combining all inherited operations with the same name so that the operation from the most specific sub-class is called first and can refer to operations from super-classes. Similarly, delegation can be viewed as combining all the operations with the same name that are linked by delegating classes. Inheritance always interprets `self` as the original target of a message even if an operation is defined in a superclass of the receiver's class. Operation lookup always finds the most specific definition in terms of the inheritance relationship but allows the next most specific definition to be referenced via `super`. Delegation should interpret `self` in the same way, but needs to provide a different keyword to allow access to the next most specific definition along the delegation relationship: `redelegate`.

We aim for both inheritance and delegation to co-exist. Since they overlap in some aspects (`self`, message lookup), but differ in others (slot storage, lookup continuation), we must choose which relationship should dominate. According to DR2 there are pragmatic reasons why giving priority to inheritance is a useful convention to handle this case.

5.2.3 Operation lookup

The first step in supporting the *delegation*-MOP is to modify all XMODELER objects to have an additional slot called `delegatee` that can be used for the delegation link. Whilst this could be done dynamically in a standard XMODELER environment,

it is easier to modify the bootstrap as follows:

```
mkClass(Object,[],[
  mkAtt("delegatee",Element,Element,false)
    ])
```

As noted above, inheritance dominates delegation. Message passing involves operation lookup based on the name of the operation and its arity. The following operation `findOp` uses `allOperations` to calculate all the operations inherited by a class, and the operation `delegatesArity` to return the number of arguments. `findOp` is supplied with the `name` of the operation, its `arity`, the `target` object, and the function `cont` which is used to continue the delegation lookup chain if necessary:

```
context Classifier
  @Operation findOp(name,arity,target,cont
    )
    // A message lookup operator that is
    used
    // as part of the new MOP. Notice that
     'cont'
    // is used to continue the lookup...
    @Find(op,allOperations())
      when op.name = name and op.
    delegatesArity() = arity
      do
        cont(op,target,@Operation()
          self.error("cannot delegate
    further.") end)
      else
        cont(null,target,@Operation()
          self.error("cannot delegate
    further.") end)
    end
  end
```

Suppose that we want to find and invoke an operation named `n` of arity `2` in a class `C` and supply arguments `vs`. If an operation exists then the value of `self` in that operation should be the object `o`. Once found, we wish to invoke the operation. Assuming that the operation always exists, this is achieved as follows:

```
C.findOp(n,2,o,
 @Operation(op,target,redelegate) op.
    invoke(target,vs) end)
```

Note that we ignore the value `redelegate` since it is not expected by a conventional operation (this will be used in a `DelegatorOp` as shown below). Note also, that the supplied value of `target` will be the same as the supplied object `o`; this will not necessarily be the case where delegation occurs as shown below.

5.2.4 Delegator definition

The meta-class `Delegator` defines a new class-level slot called `delegatesTo`:

```
context Root
  @Class Delegator extends Class
    @Attribute delegatesTo : Class end
 end
```

Note that the type of `delegatesTo` is `Class` which means that the delegation relationships can be transitive (since `Delegator` is a subclass of `Class`). The value of the `delegatee` slot is required to be an instance of the `delegatesTo` class in any instance of a role. This constraint spans two type-levels and is implemented by a mixin called `DelegatorChecker`:

```
context Root
  @Class DelegatorChecker
    @Constraint checkDelegator
      delegatee.isKindOf(self.of().
    delegatesTo)
    end
  end
```

This is added as a parent when a `Delegator` is initialised:

```
context Delegator
  @Operation init()
    let o = super()
    in self.addParent(DelegatorChecker);
       o
    end
  end
```

5.2.5 The *Delegates* meta-object protocol

The delegation-relationship affects how message passing, slot-access, and slot-update are performed in instances of roles. The corresponding meta-object protocol (MOP) implements this behavior in terms of a standard three-operation

interface in a meta-class MC. If C is an instance of MC then the MOP defined by MC defines how instances of C handle messages and slots. The operations are sendInstance for message passing, getInstanceSlot for slot access, and setSlotInstance for slot update. This Sec. defines the MOP for Delegator.

Message passing in roles is defined as follows:

```
context Delegator
  @Operation sendInstance(target,message,
    args)
    // A new MOP for Delegator that uses '
    findOp' to
    // override the default inheritance-
    based lookup
    // mechanism to use delegation
    relatinships...
    self.findOp(message,args.size(),target
    ,
      @Operation(op,target,redelegate)
        if op <> null
        then
          if op.hasProperty("redelegates")
          then op.invoke(target,args.
    prepend(redelegate))
          else op.invoke(target,args)
          end
        else super(target,message,args)
        end
      end)
  end
```

Delegator::sendInstance uses findOp to get the operation. If one exists then it is invoked otherwise super is used to continue the operation lookup. Where an operation exists, it is checked to see if it has been tagged with redelegates. In that case, it must have been defined using the DelegatorOp syntax construct (see next Sect. 5.2.6) and is expecting a special operation called redelegate as the first argument. The re-delegation argument will continue the delegation lookup as described below.

In order to support operation lookup for roles, the operation findOp is redefined as follows:

```
context Delegator
  @Operation findOp(name,arity,tgt,cont)
    super(name,arity,tgt,@Operation(op,tgt
    ,ignore)
      if op <> null
      then
        cont(op,tgt,
          @Operation()
            delegatesTo.findOp(name,arity,
    tgt,cont)
          end)
      else delegatesTo.findOp(name,arity,
    tgt,cont)
    end
  end)
end
```

The operation Delegator::findOp uses super to allow the normal rules of inheritance to find the operation. If this fails then the operation is selected by sending a findOp message to the delegatesTo class. In doing so, the value of target (eventually supplied as the value of self when an operation is invoked) is maintained. If an operation is found then it is supplied to the cont operation along with an operation used as the value of redelegate. The re-delegate operation continues the delegation by sending the delegatesTo class a findOp message, but note that the target is changed to target.delegatee so that the value of self used in any operation invoked by the re-delegation is localized.

Slot access in roles is defined as follows:

```
context Delegator
  @Operation getInstanceSlot(target,name)
    // If the slot exists in the object
    then just
    // return the value...
    if hasInstanceSlot(target,name)
    then super(target,name)
    else
      // The object delegated to may have
    the slot...
      target.delegatesTo.get(name)
    end
 end
```

Delegator::getInstanceSlot uses the method hasInstanceSlot to check whether the object has local storage for the slot that has been requested. If so then super is used to access the slot in the normal way. Otherwise, the protocol delegates the access to the delegatee. The MOP for slot access is transitive providing that delegatees are instances of Delegator.

Slot update is defined in a similar way:

```
context Delegator
  @Operation setInstanceSlot(target,name,
    value)
    if hasInstanceSlot(target,name)
    then super(target,name,value)
    else target.delegatee.set(name,value)
    end
  end
```

5.2.6 Delegator operations

Delegator operations are tagged so that they can be supplied with an additional first argument called `delegates`. This is achieved in XModeler using a new syntax construct called `DelegatorOp`. New constructs are defined by *syntax classes* whose grammar is responsible for processing the new syntax and synthesizing a performable XOCL object (see Sect. 4.2). The syntax class `DelegatorOp` is defined below:

```
context Root
  @Class DelegatorOp extends Sugar
    // The following a syntax properties
    of a
    // delegator operation...
    @Attribute name : String
    end
    @Attribute args : Seq(OCL::Pattern)
    end
    @Attribute body : Performable
    end
    @Attribute type : Performable
    end
    @Constructor(name,args,body,type) !
    end
    // The following grammar defines
    translation
    // rules from text into an instance of
     the
    // DelegatorOp class...
    @Grammar extends OCL::OCL.grammar
      DelegatorOp ::=
        n  = Name '('
        as = DelegatorArgs ')'
        t  = DelegatorOptType e=Exp 'end'
        { DelegatorOp(n,as,t,e) }
      DelegatorArgs ::=
        n  = DelegatorArg
        ns = (',' DelegatorArg)*
        { [Varp("delegates"),n] + ns }
      DelegatorArgs ::=
        { [Varp("delegates")] }.
      DelegatorArg ::=
        n = Name
        { Varp(n) }.
      DelegatorOptType  ::=
        ':' Exp
      | { [| Element |] }.
    end
    // A class that extends Sugar must
    define a
    // desugar() operation that translates
     into
    // existing syntax classes...
    @Operation desugar()
      let op = Operation(name,args,body,
    type)
      in [| <op>.setProperty("delegates",
    true) |]
      end
    end
  end
```

The `DelegatorOp` grammar processes text of a delegator-operation and produces a `DelegatorOp` instance. Since `DelegatorOp` is defined as a subclass of `Sugar`, the XModeler language processor transforms a delegator-operation into executable code by calling `desugar`. The definition of `desugar` given above returns an expression that constructs the required operation and adds a property to it.

Having defined the delegator-operation syntax construct we conclude with the definition of `delegatesArity` that must check whether the property exists on an operation in order to ignore the additional pseudo argument:

```
context Operation
  @Operation delegatesArity():Integer
    if self.hasProperty("delegates")
    then self.arity() - 1
    else self.arity()
    end
  end
```

5.2.7 Preventing cyclic delegation

To prevent cycles in delegation (SR.10), a constraint is put on `Class`. This constraint also checks for cyclic inheritance.

```
context Class
  @Constraint NoCyclicInheritance
    @Letrec hasParent(child: Class):
    Boolean =
      child.parents→includes(self)
        orelse
```

```
    child.parents→exists(p | hasParent(
  p)) orelse
    child.delegatesTo = self
      orelse
    ((not child.delegatesTo = null)
  andthen
     hasParent(child.delegatesTo))
  in
    not hasParent(self)
  end
  fail self.toString() + " has circular
  inheritance"
end
```

5.3 Comparison of the Implementation Alternatives

Delegation is an important modelling approach that is orthogonal to related approaches involving inheritance and specialization. As such, we have proposed that delegation should be supported using a dedicated language feature with associated semantics that clearly defines how structure and behavior are affected by the delegation relationship between two classes. This article has proposed operational semantics for delegation and offered two alternatives: Sect. 5.1 uses a message failure mechanism to dynamically re-target messages and Sect. 5.2 uses a meta-object protocol (MOP) to define an interface that handles references to both structure and behavior.

Redirecting messages using a failure mechanism has a number of limitations. It waits until all other lookup mechanisms have been exhausted and lead to an error being signalled. Since the error signalling mechanism is implemented in terms of a message, the handler can be redefined. However, there is little control over what has happened before the error is signalled and therefore it is not possible to allow delegation to take priority over other mechanisms such as inheritance. Furthermore, defining the operational semantics of delegation through error handling means that it is not possible to merge inheritance-based lookup with delegation-based lookup. The benefit of an error-based operational semantics for delegation is that it is easy to implement: an error message handler is redefined in those cases that a class supports delegation.

A MOP-based implementation of delegation requires an interface of operations to be defined that influences how state and operation references are performed. This approach is more flexible since, unlike the error-handling version described above, it is performed at the start of the lookup process and can take into account all the available information including delegation and inheritance relationships. A MOP typically exposes a collection of meta-operations that can be used to systematically influence the operational semantics of the underlying language. Although this will vary from language to language, it is likely to be more expressive than an error-based approach and is demonstrated in this article by showing how an MOP can be used to handle state and structure references. An obvious area for further investigation is how to use an MOP to combine multiple types of relationships (delegation and inheritance, for example).

Any modification to the lookup mechanism of a language will influence the efficiency of execution to some extent if we assume the presence of dynamic typing. An error-based approach must exhaust all possible ways of satisfying the lookup request before raising an error and thereby providing an opportunity for the new type of lookup (in this case delegation) to take control. A MOP-based approach is likely to be more efficient since it can circumvent all default lookup mechanisms at the expense of an extra test at the point of the lookup request to determine which MOP to use. In general, such a test should incur a very small overhead.

In the case of statically typed languages, performance is favoured by the error-handling mechanism. This is because the type system can verify the correctness of a reference and insert the appropriate code to call the error handler.

It is not clear whether a MOP-based approach can be integrated with static typing in order to influence its efficiency since much of the power of a MOP relies on runtime tests. However, some work can be achieved statically which relies on a type system being aware of meta-types and the MOP operations so that slot reference via

`'.'` is type-checked with respect to the meta-type of the object and can statically insert the appropriate lookup code. Some research that aims to integrate MOPs with static typing systems has been conducted by Clark (2016).

In summary, as Tab. 3 shows, there is a trade-off between the two extreme approaches demonstrated in this article. The error-based approach is much less flexible than the MOP-based approach and should be used where it is known that additional flexibility is not required. Efficiency concerns should also be taken into account, especially in dynamically typed languages, where the MOP-based approach is likely to have an advantage.

5.4 Tool Support for Delegation

When it comes to the usage of delegation at runtime, it is important to keep in mind that a new delegator instance will only be completely initialized after a corresponding delegatee instance has been assigned to its `delegatee` slot. In interactive environments, with human users creating role instances via graphical user interfaces (GUIs), it is thus required to provide interactive support to select or create the appropriate delegatee for newly instantiated delegators. In the XModeler^{ML}, interactive support for this is achieved by modifying the instantiation mechanism. If a new instance of a delegator class is created, the modified `new()` method checks at first, whether instances of the corresponding delegatee class exist. If that is the case, the user is offered a selection list from which one of these instances can be selected to act as delegatee for the new instance. The selected object is then assigned to the instance variable `delegatee` within the previously created delegator object. If there is no available instance of the required delegatee class in the system, a dialog is shown that informs the user that an instance of the delegatee class is required. The user can then decide whether to roll back the instantiation, or whether a new instance of the delegatee class should be created, which would then be linked to the delegator object.

Fig. 19 shows the diagram editor of the XModeler^{ML}. The screenshot shows a multi-level diagram containing classes and objects on levels M1 and M0. Following a part of the example in Fig. 5, two classes have been created. One of them, `Employee`, delegates to the other, `Person`. Three objects of `Person` have also been created. When the user now creates a new `Employee` object, delegation demands that an object of `Person` is supplied to ensure that the new `Employee` object has a target to delegate to. This is handled via the displayed selection dialog in the above-described way.

6 Related Work

The concept of delegation is discussed with multiple facets in the existing body of scientific literature. Several approaches examine the notion of delegation exclusively on a conceptual level and in the context of conceptual modeling (Chu and Zhang 1997; Gamma et al. 1994; Lieberman 1986; Steimann 2000a,b; Stein 1987).

An additional perspective is taken in when delegation is understood as a tool for software modeling and system construction (Jäkel 2017; Kühn et al. 2014; Leuthäuser and Aßmann 2015; Riehle 2000), which is also the primary perspective from which our work looks on delegation.

Delegation and roles also appear as constructs in programming languages. Several different options for realizing their formal semantics are explored with the help of programming language implementations (Dony et al. 1998; Prototype-Based Programming 2001; Stefik and Bobrow 1986; Zivkovic and Karagiannis 2016).

A number of representatives from each of the aforementioned categories are discussed in the upcoming sub-Sect.s.

One should be aware that, as the term “delegation” is used in a wide variety of works, there is no unified understanding of it. This holds true also for the notion of the relationship between delegation and the use of roles. Some approaches regard delegation and the use of roles as inherently connected to each other, like also our approach does. For others, delegation is a kind of relationship between objects and their classes, which does not require to talk about the notion of roles.

Table 3: Comparison of the implementation alternatives

Implementation	Message Failure (see Sect.)	Meta-Object Protocol (see Sect.)
Ease of modification	⊕ *high*, regular programming constructs are used	⊖ *low*, internal language execution mechanism is modified
Flexibility	⊖ *low*, conditions under which a message failure is thrown cannot be controlled	⊕ *high*, fine-grained control over priorities among inheritance, delegation, re-definitions
Performance	⊖ *low*, exception handling mechanism adds overhead	⊕ *high*, integrates with existing method lookup mechanism without additional overhead

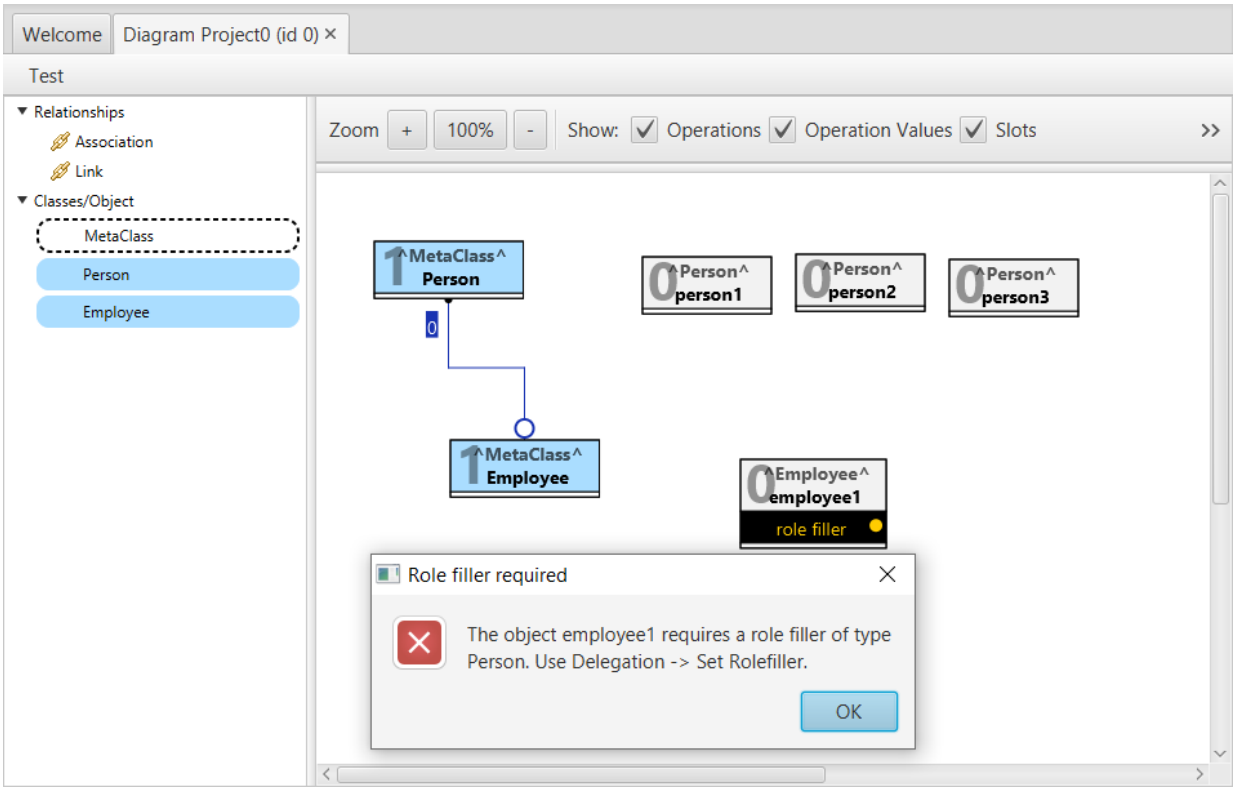

Figure 19: Dialog shown on instantiation of a role

6.1 Conceptualizations of Roles and Delegation

Early conceptualisations of delegation have been proposed in Lieberman (1986). In that work, essential philosophical distinctions are made between a knowledge representation approach based on classes, and one based on prototypes that can make use of delegation. The term "delegation" appears to first have been introduced in that work as a concept for software architecture specification. However, delegation is solely discussed as a design element for class-less, prototype-based languages.

A fundamental difference in how roles are understood can be noticed with regard to the question whether roles are considered to be independent entities of their own kind, i. e., whether they have their own identity, or whether roles are understood as to be taken in by objects, i. e., they represent

modes of operation that objects can have. An understanding of roles as first-class concepts in their own right is suggested by Chu and Zhang (1997), which argues for incorporating the language elements "role" and "association" into the set of object oriented system design metaphors, together with classes, attributes, and references. This approach is contrary to our conceptualisation, which uses the delegation relationship as additional language element to constitute the use of classes as role classes and / or role filler classes. The relationship between roles and their "players", which is how role fillers are called in Chu and Zhang (1997), is not discussed in-depth in that work, because the examination primarily focuses on the use of roles as participants in associations. Also, realization options for implementing delegation in programming languages are not mentioned.

6.2 Roles and Delegation in Conceptual Modeling

A proposal for conceptualising delegation as part of object-oriented languages is given in Frank (2000). That work explains the semantics of delegation according to our approach, using fundamental terms such as "role" and "role filler". It can thus be regarded as one foundational predecessor of this article. Sketches for implementing delegation both as part of a modeling language, as well as in the programming language Smalltalk, are included as well.

A special focus on delegation is taken in when understanding it primarily as a mechanism for encapsulation and reuse. This brings delegation close to the notion of inheritance and leads to the question how both compare to each other. Accordingly, Selic et al. (1994), Stein (1987), and Strahringer (1998) and others examine the specific relationship between delegation and inheritance, and come to the conclusion that delegation is semantically overlapping with inheritance and allows to reflect all features offered by inheritance as well. As our analyses have shown, this view does not account for the complete picture, because with respect to expressing common characteristics of a group of objects, abstract classes offer additional expressiveness and allow polymorphic constructs which cannot fully be reflected by delegation (see Sect.s 2.2 and 3.1).

Cook (1989, 1992) and Cook et al. (nodate) also examine options for achieving reusability of behavior definitions, and state that "Inheritance Is Not Subtyping" (Cook et al. nodate). Most programming languages, the type-safe ones in particular, define inheritance in a way that it satisfies subtyping. While the motivation of subtyping is to ensure substitutability among classes, the definition of inheritance in Cook et al. (nodate) is a concept which supports the reuse of implemented functionality. The latter definition broadly matches our definition of delegation. The use of one concept to satisfy both motivations, as many programming languages do, likely leads to inconsistencies, mostly around co- or contravariance issues. Consider natural and rational numbers for instance. With a focus on substitutability the natural numbers can be used where rational numbers are demanded, so the natural numbers were a subclass of rational numbers. With a focus on code reusability, the operations of the natural numbers are reused by the rational numbers, so the rationals were the subclass.

Riehle (2000) has a different view on delegation. That work examines ways to construct system behavior out of independently defined parts, and regards delegation as an alternative mechanism to inheritance for composing system behavior out of separately defined elements. In contrast to our approach, a class can be explicitly defined as being able to fill a role. Whereas in our approach a class is not aware of its delegatee qualities, comparable to a superclass not being aware of its subclasses. In our approach a delegator is like a conventional class, but requires a delegatee. This allows the delegatee to temporarily act as the delegator, which would not be possible with inheritance where such changes over time are not possible. The approach of Riehle (2000), on the contrary, has an inverted direction of dependency. There, a class can be composed of roles without possibilities for any temporal variation. This approach therefore clearly misses out on requirement DR 5. Instead,

the motivation is to use inheritance to add the description of properties to classes, and roles to add the description of behavior.

In Steimann (2000a,b), the idea of objects playing roles is promoted in the context of a general examination on object collaboration, polymorphism, and substitutability. Those elaborations accordingly conceptualize roles as individual entities. An extensive contribution about the conceptualization, application and implementation of roles is provided by Steimann (2000a), where a synthesis of diverse role conceptualizations that are present throughout the related literature is suggested. An essential standpoint of this work is that roles necessarily represent an orthogonal modeling concept with a meta-type of its own, and that "an undifferentiated combination of roles and natural types in one substitution hierarchy almost necessarily leads to a certain randomness of design, and to confusion of the viewer" (translated from German) (Steimann 2000a, p. 79). As part of its elaboration, Steimann (2000a) identifies a set of characteristics that describe the notion of roles represented by that work. Among them are, e. g., "A role comes with its own properties and behavior", "An object may play different roles simultaneously", and "Roles can play Roles" (Jäkel 2017). The notions expressed with these characteristics widely resemble the understanding of roles as it is present in our work.

An additional aspect is added to the understanding of roles by Kühn et al. (2014). That approach understands roles as concepts that come into existence not only when being played by objects, but additionally by being embedded into a context that provides structure and gives meaning to the role. Without this context, called "compartment" in Kühn et al. (2014), the role remains meaningless. An example is the context of a "university", which is constitutive for the roles of "professors" and "students". These roles require to be located in the context of "university", in order to be reasonably played by acting entities.

In our approach, we do not regard it as necessary to consider the context explicitly, because a sensible definition of a role will necessarily imply domain relations to contextual elements, in order to define the role's purpose in the domain of discourse.

6.3 Roles and Delegation in Software Modeling

Szyperski et al. (2002) also studies differences and commonalities among delegation and inheritance, with a strong focus on the consequences that arise for the implementation of object behavior. The work differentiates between interface inheritance, which allows for substitutability of superclasses by subclasses, and implementation inheritance, which allows to reuse behavior definitions of superclasses by objects that instantiate a subclass. Like other representatives of scientific work on delegation, Szyperski et al. (2002) comes to the conclusion that "object composition shares several of the often quoted advantages of implementation inheritance" (Szyperski et al. 2002, p. 133). To further differentiate between implementation inheritance and message forwarding, dynamic aspects of object composition are investigated in detail. Especially the fact that implementation inheritance jeopardizes the predictability of program control flow when objects invoke methods of their own class that have been overwritten by subclasses is considered a serious problem. On the one hand, it is typical for the implementation of object behavior that methods that belong to the same class invoke each other, which allows to decompose object behavior in a structural way to make it better understandable. On the other hand, the resulting network of self-reflexive, re-entrant method invocations becomes incontrollably complex when individual methods are overwritten by subclass definitions, which even may branch back into the original "super"-implementation of methods in order to compose their new behavior. Unforeseeable recursions can be the result of this. As Szyperski et al. (2002) concludes, this danger emerges from the availability of a "common self" when implementation inheritance is used. In contrast to this, composed objects that invoke each other

without implementation inheritance avoid the danger of uncontrollable recursion. As a conclusion, the work summarizes that "[t]he combination of object composition and forwarding comes fairly close to what is achieved by implementation inheritance. However, it does not get so close that it also has the disadvantages of implementation inheritance" (Szyperski et al. 2002, p. 134). This notion of "object composition and forwarding" resembles what most representatives of the scientific literature, as well as this article, call delegation. Delegation in the sense of Szyperski et al. (2002) extends this idea of message forwarding by introducing an option for using a "common self" in combination with message forwarding, however, not without mentioning that this can make "object composition as problematical as implementation inheritance" (Szyperski et al. 2002, p. 135).

Jäkel (2017) applies the concept of roles to database modeling and data management. During the discussion of foundations, the work points out that, according to Kühn et al. (2014), there are three perspectives from which role conceptualizations can be categorized. These are the *relational perspective* ("different entities interact with each other or are connected by using roles"), the *structural and behavioral* perspective (attributes and methods can flexibly be assigned to entities during runtime), and the *context-dependent* perspective ("roles are utilized to describe context dependent features of entities") (Jäkel 2017, p. 14). Each combination of perspectives is discussed on the background of existing approaches that represent this combination. Our approach covers all three perspectives. Taking also the *compositional* perspective of Riehle (2000) into account, the set of four perspectives spans a conceptual framework by which the conceptualizations of roles and delegation of each publication can be categorized.

Another option to incorporate roles into executable software is to offer integrated modeling and implementation perspectives onto roles and delegation relationships that are inherently consistent with each other. The XModeler^{ML}(see Sect. 4.1) allows to represent alternative views onto the same internal representations of roles and delegation relationship declarations. Thus it allows to specify roles in models, which are at the same time internally represented as executable program constructs, without the need for explicit code generation from the model. This inherently prevents inconsistencies between the specification of roles and their implementation through the shared representation.

Consistency between the declaration of roles and delegation relationships in a conceptual model, and their realizations as part of a running software-system, is an important prerequisite to apply the concepts of roles and delegation to software development. If roles and delegation relationships are specified using a conceptual modeling language with suitable tool support, there are two basic options for deriving corresponding implementations. One option is to make use of a shared representation of the model and internally executable language elements, as in XModeler^{ML}(see above). Another way is to apply code generation techniques that generate implementation code, potentially for multiple target programming languages. In the case of using code generation, the correct implementation of declared roles and delegations depends on the correctness of the according code generation templates. Kühn (2017) provides a modeling language for conceptualizing roles, together with a corresponding modeling tool. For making the created models executable, the modeling tool encompasses a number of code generators. One code generator is especially worth mentioning as it generates source in SCROLL (SCala ROLes Language) (Leuthäuser 2015; Leuthäuser and Aßmann 2015) (see 6.4.1).

6.4 Roles and Delegation in Programming Languages

While prevalent programming languages usually do not support delegation, research has produced various programming languages that offer concepts such as roles or delegation.

6.4.1 Role-based programming languages

Role-based programming languages provide language constructs as first-class citizens for declaring roles and their use in object-oriented programs.

SCROLL (Leuthäuser 2015; Leuthäuser and Aßmann 2015) is implemented in Scala (Odersky et al. 2016) and provides dedicated support for roles. Despite being a statically typed language, Scala allows for dynamic message dispatching. SCROLL utilizes this for implementing a message dispatching similar to our conception of delegation. In SCROLL an object may have roles, which can be activated during runtime. While a role is active, the object's behaviour changes, like a person changes his / her style of talking once stepping on a stage. E. g., when a *Person* object is asked to *talk()*, the runtime engine recognises the person being in the compartment (Kühn et al. 2014) *Stage* and invokes the *talk()* operation of the *Actor* role. However, in contrast to our approach, there is no separate *Actor* object with its own identity in that approach. When using delegation in the way proposed by our work, both delegator and delegatee have their own identity.

JAWIRO (Java with Roles) (Selçuk and Erdoğan 2004, 2006) is an extension to the Java language. It shares some requirements with our approach, amongst others that delegatees can be delegators themselves in another relation (SR9), and that the relationships between delegators and delegatees may change during their lifetime (DR5). JAWIRO strongly focuses on implementation and on runtime efficiency, whereas our approach focuses on a clean conceptual representation to reconstruct natural language domains. To support JAWIRO's focus on efficiency, much of the role related implementation needs to be added manually each time the concept is used, whereas our approach simplifies the implementation by having delegation built in as a base concept of the programming language.

Another Java-based language which operates with the notion of roles is Object Teams (Herrmann 2003). In Object Teams, roles appear solely as means for modularization, without any delegation involved. In the center of the approach lies the extension of the notion of a `package` to the notion of a `team`. While packages in Java provide merely a grouping of classes that affects the visibility of internal class members among the classes in the same package, the semantics of teams allows for a more powerful control over members in the team, and it provides an abstraction over the idea of a collaboration among objects, by allowing teams to have their own states and behavior. Objects in teams are defined by *roles* which are classes defined inside the team. Roles can be attached to regular classes using the `playedBy` keyword. The semantics of a role being attached to a class is that it modifies selected aspects of the class's behavior, using mechanisms known from aspect-oriented programming (see Sect. 6.4.4). In this sense, roles in Object Teams can be understood as bundles of modifications to class behavior which each are defined as aspects of the modified class. Roles are collections of aspect-oriented modifications that influence the behavior of role instances in a team. This notion of roles is significantly different from most other approaches that use the same term, including ours. At first, this is because the notion of roles is not interrelated to a conceptualizaiton of delegation at all. Secondly, attaching roles to classes is performed on the type-level and happens at build time, which prevents the notion of a role from dynamically reflecting aspects of an object's lifecycle.

A role-based extension to Java is also provided by PowerJava (Baldoni et al. 2006). As in Object Teams, PowerJava understands roles as constituting parts of a surrounding context, this time called "institution". Also, the language extension mechanism that is provided for role support exclusively operates on the type-level by extending the original Java semantics with a sort of polymorphic method dispatcher that can branch into different implementations of a method depending on the type with which an object is used (Baldoni et al. 2006). Like Object Teams, PowerJava thus exclusively uses roles as structural declaration elements. In PowerJava, the focus primarily lies on providing more fine-grained declaration elements for an object-oriented type system, while in Object Teams, the abstraction of a collaboration is at the forefront of the intentions behind the language. A few commonalities with our approach are found on the implementation

level when it comes to role-dependent method dispatching. Conceptually, the previously discussed approaches widely differ from our approach, as well as from any approach which incorporates a notion of role-lifecycles and dynamic assignment of roles, and regards roles as conceptual elements to constitute an overall notion of delegation.

6.4.2 Prototype-based languages

Delegation principles are not only considered in combination with role-based approaches, but are also used as a foundational abstraction mechanism of prototype-based languages (Lieberman 1986). Such languages provide an object-oriented view on systems without relying on the notion of classes as constituents for the existence of objects, while ensuring a level of semantic expressiveness equal to class-based languages (Prototype-Based Programming 2001). In programming languages that make use of classes, inheritance can serve as a language-inherent abstraction mechanism for encapsulating and re-using object behavior that addresses general aspects of a group of objects. This mechanism is not available in prototype-based languages, in which groups of objects with common characteristics are created as instances that reference a common prototype object. Method calls and read-accesses to properties on each of the individual objects will then be delegated to the prototype object, if they reference the common characteristics defined by the prototype. As Dony et al. (1998) shows, by factoring out common behavior of a group of objects into such a "super"-prototype, and delegating method calls to this commonly shared prototype, the same semantic expressiveness as provided by inheritance can be made available in prototype languages.

A prominent representative of prototype-based languages is JavaScript Flanagan 2011. In JavaScript, each object owns a reference `prototype` to another object, which is evaluated by the language execution mechanism as the transparent target for delegation, if methods or properties are accessed on an object which are not made available by the object itself. This kind of delegation provided by prototype-based programming languages does not make use of roles, since the delegation is intended to happen transparently between objects. It is thus to a wide extent different from the delegation approach we follow, which is intended to provide an explicit means of abstraction for delegation relationships in a domain model.

Bettini et al. (2003) distinguishes between *consultation* and *delegation* which differ in their target of the self-reference, usually named `self` or `this`. In their definition of *delegation* the self-reference refers to the object invoking an operation. This concept is commonly used by prototype-based languages as it can be seen as comparable to inheritance of operations in class-based languages. For their definition of *consultation*, the self-reference refers to the object which is supplying the operation. The latter concept corresponds to our definition of delegation.

6.4.3 Distributed programming languages

A special notion of delegation without roles is realized by distributed programming languages, i. e., programming languages that allow the development of distributed systems based on a location-transparent language paradigm. A representative of such a language is Emerald (Raj et al. 1991), which is a class-less object-oriented language that also incorporates language mechanisms to deal with the location and mobility of objects (Black et al. 1986). Emerald's language design allows to expose methods of objects to make them accessible from other objects, independent from where both objects are physically located in a distributed system. To realize such a distributed architecture, the internal mechanisms of the language's runtime engine needs to keep track of the location in which objects reside. Depending on whether accessed objects are local or remote, the invocation mechanism can transparently decide whether a local method call is realized, or whether a remote invocation via a local proxy is performed. In the latter case, one could speak of a method being transparently delegated from a local proxy object to a physically remote object. This kind of delegation intentionally does not make use of roles, because

it is the very purpose of the language to hide the internals of the delegation that takes place, in order to achieve location-transparency. The question whether an object is actually accessed on a local system, or an object is playing the role of a proxy which delegates the actual program behavior to a remote object, remains transparent from the programmer. The programmer does not have to take care of distribution characteristics and thus can remain in a consistent mindset while developing scalable applications that can be developed and tested locally and later transparently be distributed (Black et al. 1986). On the one hand, this provides a less error-prone way of developing distributed systems, because the view taken in on a system by the programmer is conceptually cleaner and not interwoven with management functionality to handle local distribution. On the other hand, this transparency may lead to unexpected program behavior regarding non-functional characteristics of software systems, especially with respect to performance issues caused by slow network connectivity, or deadlocks that may occur due to dysfunctional network communication. In distributed programming languages, programmers do not have control over the mechanisms that handle local distribution. Because of this implicit nature of role-less delegation in distributed programming languages, this use of delegation differs significantly from role-based delegation approaches, including ours, which aim at explicating delegation relationships intentionally.

6.4.4 Aspect-oriented programming

The idea of using aspects (Elrad et al. 2001; Steimann 2006) for defining program behavior shares a number of characteristics with delegation conceptualizations. Aspects represent pieces of behavior which are defined separately from other program units, and subsequently are "woven" into other behavior definitions. This resembles the idea that at some points, a program's behavior should not only consist of definitions made particularly for one method or function, but in parts the responsibility for executing the program is shifted to elements that are factored out of the regular behavior definitions. In this sense, responsibility for behavior execution is "delegated" to aspects. This approach provides a definition mechanism which allows to compose behavior flexibly from independently declared elements, which is in line with our conceptualization of delegation, and also resembles fundamental notions by Kühn et al. (2014), Riehle (2000), and Steimann (2000a) and others. A difference to our notion of delegation is that aspects assume a static composition of behavior at build time, while especially for our approach, dynamic assignment of behavior during runtime is important to be able to reflect a range of domain characteristics which can change over time.

6.4.5 Interfaces, mixins, and traits

The use of *interfaces* in object-oriented system specification also shows a number of similarities with the use of roles. Like roles, interfaces provide a partial definition of an object's behavioral capabilities, rather than demanding for covering the entire range of an object's behavioral features. It is also inherent to the notion of interfaces that they can be combined to compose the description of object behavior from several parts that have originally been defined independently from each other (Steimann 2001). While these similarities refer to the capabilities of interfaces and roles to contribute to an object's method signature, i. e., to the question in which contexts an object can be used, the question of how this behavior is implemented, and possibly reused from existing implementation specifications, is addressed differently by interfaces and roles. Roles allow to factor out the description of behavior in the form of method signatures, and also provide a mechanism for defining the actual implementation of behavior separate from class definitions. In addition, they come with defined semantics on how to include their behavior into an object's behavior, e. g., by using delegation as suggested by approaches as ours. This aspect of implementation reuse, which is also stressed by Riehle (2000), constitutes a substantial difference between interfaces and roles, regarding roles not only as passive facets through which

objects can be accessed, but also as elements for the modularization of behavior specifications.

Mixins (Stefik and Bobrow 1986; Zivkovic and Karagiannis 2016) are another approach with similarities to roles, especially if roles are understood as means for achieving modularization. Stefik and Bobrow (1986) describes mixins as "special classes that bundle up descriptions and are 'mixed in' to the supers lists of other classes in order to systematically modify their behavior". If a mixin is added to a class, its features are added to the class in addition to the features it inherits from its superclass. Mixins are compositional entities on a finer level of granularity than classes. They represent parts of class functionality, that can be shared among multiple classes, however, they are not intended to be instantiable classes on their own. This makes them differ from concpetualizations of roles which indeed ascribe separate identities to roles.

Being incomplete entities that cannot exist on their own is a shared feature between mixins and roles in delegation. For mixins, however, this dependency is effective only on the class level at build time, while delegator in our understanding additionally require their delegatee objects to be associated at runtime.

There are two common options to implement mixins. On the one hand, languages can treat mixins like classes to inherit from them. On the other hand, in languages such as Ruby (Flanagan and Matsumoto 2008; Ruby Programming Language nodate) or Scala (Leuthäuser and Aßmann 2015), mixins are independent language concepts that can be included as bundles of functionality. In either case, a mixin is not instantiated by its own, but only as a part of the class it has been added to (Stefik and Bobrow 1986).

Besides mixins, *traits* are yet a further option for enhancing a class with functionality. Schärli et al. (2003) defines a trait as a "set of methods that implement [...] behavior". Traits are thus a special form of mixins. They are also inserted as a bundle of functionality into a class, but traits are not self-contained. When they are inserted, they need to be parametrized with behavioral or structural features of the containing class. Both the concepts of mixins and traits appear in various forms in the literature. To achieve a more precise understanding of the terms, further research is required.

The Java based programming language Groovy (Knig et al. 2015) uses traits. Similar to our approach method invocations are delegated, in this case from a class to a trait. The trait can have fields and therefore have a state. In contrast to our approach, the trait does not have a separate identity like a delegatee. It is part of the delegating class and cannot exist without it. Any dynamic changes to the delegation relation are impossible without an identity.

6.4.6 Delegation versus inheritance in object-oriented programming languages

Orrù et al. (2015) and Tempero et al. (2013) conducted an empirical study on the usage of inheritance in object-oriented programming languages. They examined open source code from various projects written in Java and Python and tried to decide what the motivation behind making use of the inheritance relation was. A significant number of cases was found where inheritance should better be replaced by, e. g., composition. Bloch (2008) defines composition as having a field pointing to an object of the class where reused functionality comes from. This approach resembles our delegation pattern, albeit it is not built-in and needs wrappers to be added manually.

More recent object-oriented programming languages, such as Rust (Matsakis and Klock 2014) and Go (Donovan and Kernighan 2015), do not make use of classes, but rather allow to define types as compositions of other types. As a consequence, they also do not make use of the notion of inheritance, but instead provide advanced means for interface composition.

The development of Rust was mainly motivated by the wish to overcome deficiencies of the C and C++ languages, while still targeting a low-level, hardware-related execution environment. In the first place, this means that Rust,

like C and C++, is a language which compiles to hardware-dependent machine code, but it provides a safe memory management by default which avoids typical problems such as buffer overruns and dangling pointers. Besides such general improvements, RUST also offers a type abstraction system which makes use of *traits* as fundamental abstractions to factor out structural and behavioral commonalities from individual types to common descriptions over a group of types (Matsakis and Klock 2014).

As part of its traits implementation mechanism, the RUST compiler is able to handle type polymorphism in two equivalent, but fundamentally different implemented, ways. At first, it can compile various versions of operations for different concrete types in parallel, which realizes a zero abstraction overhead for the generated code. Alternatively, operations can be compiled as accessing traits generically, which requires to offer a dynamic mode of accessing them via an indirection mechanism that resembles one form of delegation (Turon 2015). Here, delegation is used as an internal implementation alternative to static composition of behavior, in order to implement higher-level type abstractions. This kind of delegation does not become visible to the user of the programming language and is not expressed explicitly by the programming language.

One could argue that an understanding of delegation in this sense represents the starting point of a continuum, which ranges from entirely implementation-driven approaches that boil down to dynamically resolving a function address reference, to fully conceptually driven approaches which understand delegation and related concepts as description means to express concepts of a domain. Our approach resides at the latter end of this continuum.

In GO, as another example, a type can be defined by specifying a structure with attributes and methods in a traditional way, and in addition, by embedding other types into the type definition. This way, all characteristics of embedded types are made available to the embedding type as well, which allows the embedding type to be composed of any number of existing types. In the terminology of the GO language specification, this is achieved by *promoting* the characteristics of embedded types to the embedding type (Donovan and Kernighan 2015). In other words, accesses to attributes and methods that are not declared as individual members of an embedding type are transparently delegated to embedded types. This notion of delegation also does not rest upon the use of roles, it does not even demand for the declaration of interfaces for the embedded types (interfaces are never explicitly declared in GO, every type that implements the methods of an interface is automatically treated as being able to satisfy that interface).

Like in RUST, the use of delegation in GO thus also is restricted to internal language processing mechanisms, and rather serves implementation purposes than provides meta-concepts for describing object-oriented systems. While from the point of view of our approach this represents the other side of the conceptual continuum mentioned above, it seems reasonable to incorporate these implementation mechanisms into a comprehensive examination of delegation approaches, to cover the full spectrum of conceptual notions around the term “delegation”.

6.5 Delegation as Design Pattern

A view which mostly focuses on constructive aspects of designing systems comes into play by suggesting the use of delegation as a design pattern for system construction. Gamma et al. (1994) discuss the use of the delegation pattern specifically to promote this aspect. In their collection of design patterns, delegation is incorporated, but merely reduced to a mechanism for replacing missing inheritance semantics. While delegation is indeed capable of partially replacing the inheritance semantics as discussed in Sect.s 2.2 and 3.1, the narrow perspective of Gamma et al. (1994) does not fully account for the conceptual capabilities of letting objects play the role of other ones, interchangeable at runtime. In addition to mentioning delegation as “a way of making composition as powerful for reuse as inheritance” (Gamma et al.

1994, p. 31), Gamma et al. propose two patterns, "Proxy" and "Chain of Responsibility" that recommend transparent message forwarding similar to delegation. However, both patterns are mainly intended for certain technical purposes and lack a conceptual foundation comparable to delegation.

A special case of using delegation between objects is the use of the "Builder" pattern (Gamma et al. 1994). A Builder is an object which serves to instantiate other objects, comparable to the "Factory" design pattern (Gamma et al. 1994). It is mostly used in cases when objects are created that demand for a complex configuration to be fully instantiated, for example, a large number of attributes that need to be set before the object can be used. Instead of creating such objects directly by instantiating their class and then set the attributes as desired, using a separate Builder object allows to first configure all attributes, and then create the desired object and internally pass all previously set attributes to that object. A Builder thus provides a kind of deferred delegation: the Builder manages all required attributes and allows to set them in a way similar to setting them directly on the object to be built. But instead of providing functionality that processes these attributes, they are stored for the purpose of passing them on to a new object to be built. In this sense, the Builder serves as a temporary memory of the configuration of objects, and delegates the configuration is has received to new objects. Besides a temporal decoupling of the configuration process of an object and its instantiation, this pattern also allows to apply the same configuration to multiple object instances, thus to create any number of objects that are equally configured, while the Builder only needs to be configured once. This use of delegation is different from our proposed approach, not only because it operates without the notion of roles, but because it intentionally mirrors object characteristics for the purpose of providing proxy capabilities that allow the Builder to act as a placeholder for objects that are created at a later point in time.

7 Conclusions and Future Work

The presented work has conceptualized a notion of delegation that extends the existing set of language elements for object-oriented system design and implementation. We have discussed various options for realizing delegation in object-oriented specifications, and have compared the use of delegation to the use of inheritance. It satisfies all requirements presented in Sect. 2.3. With a list of guidelines for using delegation, the article contributes to its practical application, and to a pragmatic understanding of delegation as one additional means of expression for describing object-oriented systems. By providing two alternative implementations of delegation in the XMODELER language engineering environment, prototypes have been created, which exemplify the potential of delegation both as a modeling construct and as part of an underlying object-oriented programming language. With these implementations available, our aim to provide a language framework for delegation within which the different approaches can be constructed, analyzed and compared, is fulfilled.

Note that the current implementation in the XMODELER accounts for all aspects of the specification. Out of the two implementations we presented in this paper, we decided for the one where "self" refers to the delegator object, because it is more appropriate from a conceptual point of view. Also, the implementation in the XModelerML does not support the optional requirement SR3 (and, as a consequence, DR5), because in most cases it is more appropriate to restrict the number of simultaneous delegatee objects to one. As a consequence of DR3 delegation to class (MR2) is not included in the current version of the XMODELER . Its implementation would, however, be straightforward using the already existing implementation of delegation between objects at the same level.

The elaborations in this article provide the foundation for diverse further directions of research. The realization available so far is based on a dynamically typed programming language with type-checking at runtime only. Integrating delegation

semantics into the grammar of a statically typed language, i. e., a language that allows for checking type conformance by validating the source code at development time, would allow for ensuring valid assignments of delegatees to delegators at development time, and could detect ambiguities and name clashes in signatures, e. g., when the same method signature is inherited and delegated. Incorporating these features into a statically typed language is thus one of the next topics to focus conceptual research on.

The idea of delegation can be extended from referring to behavioural features only, to also include structural features. This means that attributes of classes can be made accessible via delegation as well. In such a setting, different modes of operation can be distinguished, e. g., delegation in cases when no own attribute declaration is available to the delegator, or delegation for the purpose of providing a default value, in cases when an attribute is empty or has a `null` value. Questions of encapsulation and attribute visibility may additionally have to be reconsidered when delegation is added to the available modes of accessing attributes.

The potential use of delegation in multi-level type systems (Frank 2014) also raises a number of new research questions. How should delegation across multiple levels of a class hierarchy be understood, and should it generally be allowed for in a multi-level type hierarchy? What are the consequences of combining delegation and inheritance, and what implications go along with covariant / contravariant refinements of classes that stand in relation to other classes via delegation? For example, a class `Role1` delegates to the class `Rolefiller1`. Class `Role1A` is a subclass of `Role1`. Then `Role1A` requires a delegatee of class `Rolefiller1`. Class `Role1A` could now be altered to delegate to `Rolefiller1A`. What can we say about the relationship between `Rolefiller1` and `Rolefiller1A`? Would substitutability require an object of `Role1A`, which can act as an object of `Role1`, to accept any delegatee of `Rolefiller1`? Such questions are to be addressed in subsequent work.

To support programming with delegation, reflexive language capabilities for introspecting delegators, delegatees, and delegated operations, are yet to be provided. Such functions not only increase the degree of programmatic control over the delegation mechanism but can rather serve as debugging instruments when it comes to understanding the runtime behaviour of objects that are involved in delegation relationships.

On the level of modeling pragmatics, implications for the usability of delegation in modeling and programming languages need to be examined. One central question is how to represent behavioural elements that are made available to a delegator through a delegation relationship, and, vice versa, how to indicate by a delegatee, which elements are intended to be made available via delegation.

We are convinced that this work substantially contributes to a wider and more reflected use of delegation, both in conceptual modeling, as well as in software engineering, and in particular in Information Systems. We hope that the presented results foster a wider use of delegation because it is clearly suited to improve reuse, flexibility and integrity of software systems.

The use of delegation is demonstrated in a screencast at https://le4mm.org/delegation/. The corresponding models as well as the XModeler^{ML}are available at https://le4mm.org/xmodelerml/#download.

References

Ambler S. W. (2004) The object primer : agile modeling-driven development with UML 2.0, Third edition.. Cambridge University Press

Ambler S. W. (2005) The elements of UML 2.0 style eng. Cambridge University Press

Atkinson C., Gerbig R. (Jan. 1, 2016) Flexible Deep Modeling with Melanee. In: Reimer S. B. U. (ed.) Modellierung 2016, 2.-4. März 2016, Karlsruhe – Workshopband Vol. 255. Gesellschaft für Informatik, pp. 117–122 published

Atkinson C., Kühne T. (2001) The Essense of Multilevel Metamodeling. In: UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. Lecture Notes in Computer Science Vol. 2185. Springer, pp. 19–33

Atkinson C., Kühne T. (2008) Reducing accidental complexity in domain models. In: Software & Systems Modeling 7(3), pp. 345–359

Ayesh A. (2002) Essential UML fast: using SELECT use case tool for rapid applications development eng. Springer

Bachman C. W., Daya M. (1997) The Role Concept in Data Models. In: Proceedings of the Third International Conference on Very Large Data Bases - Volume 3. VLDB '77. VLDB Endowment, Tokyo, Japan, pp. 464–476

Baldoni M., Boella G., van der Torre L. (2006) powerJava: Ontologically Founded Roles in Object Oriented Programming Languages. In: Proceedings of the 2006 ACM Symposium on Applied Computing. SAC '06. ACM, Dijon, France, pp. 1414–1418

Balzert H. (2011) Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2 ger, 2. Aufl., Nachdr. Spektrum Akad. Verl.

Bettini L., Capecchi S., Venneri B. (2003) Extending Java to dynamic object behaviors1 1This work has been partially supported by EU within the FET - Global Computing initiative, project AGILE IST-2001-32747 and by MIUR project NAPOLI. The funding bodies are not responsible for any use that might be made of the results presented here. In: Electronic Notes in Theoretical Computer Science 82(8) WOOD2003, Workshop on Object Oriented Developments (Satellite Event of ETAPS 2003), pp. 33–52

Black A. P., Hutchinson N. C., Jul E., Levy H. M. (1986) Object Structure in the Emerald System. In: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, Proceedings., pp. 78–86

Bloch J. (2008) Effective Java (2Nd Edition) (The Java Series), 2nd ed. Prentice Hall PTR

Cardelli L., Wegner P. (1985) On Understanding Types, Data Abstraction, and Polymorphism. In: ACM Comput. Surv. 17(4), pp. 471–523

Chu W. W., Zhang G. (1997) Associations and roles in object-oriented modeling In: Conceptual Modeling — ER '97: 16th International Conference on Conceptual Modeling Springer, pp. 257–270

Clark T. (2016) Static meta-object protocols: towards efficient reflective object-oriented languages. In: Fuentes L., Batory D. S., Czarnecki K. (eds.) Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016. ACM, pp. 160–167

Clark T., Gonzalez-Perez C., Henderson-Sellers B. (2014) A foundation for multi-level modelling. In: Proceedings of the Workshop on Multi-Level Modelling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014), Valencia, Spain, September 28, 2014., pp. 43–52

Clark T., Sammut P., Willans J. (2008a) Applied Metamodelling: A Foundation for Language Driven Development

Clark T., Sammut P., Willans J. (2008b) Superlanguages: developing languages and applications with XMF. Ceteva

Clark T., Willans J. (2012) Software Language Engineering with XMF and XModeler. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments. IGI Global, pp. 311–340

Coad P., Yourdon E. (1991) Object oriented design eng. Yourdon Press

Cook W. R. (1989) A Proposal for Making Eiffel Type-Safe. In: ECOOP '89: Proceedings of the Third European Conference on Object-Oriented Programming, Nottingham, UK, July 10-14, 1989., pp. 57–70

Cook W. R. (1992) Interfaces and Specifications for the Smalltalk-80 Collection Classes. In: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92), Seventh Annual Conference, Vancouver, British Columbia, Canada, October 18-22, 1992, Proceedings., pp. 1–15

Cook W. R., Hill W. L., Canning P. S. (nodate) Inheritance Is Not Subtyping. In: Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990, pp. 125–135

Donovan A. A., Kernighan B. W. (2015) The Go Programming Language, 1st ed. Addison-Wesley Professional

Dony C., Malenfant J., Bardou D. (1998) Classifying Prototype-based Programming Languages. In: Prototype-based Programming: Concepts, Languages, and Applications

Elrad T., Filman R. E., Bader A. (2001) Aspect-oriented Programming: Introduction. In: Commun. ACM 44(10), pp. 29–32

Embley D. W., Kurtz B. D., Woodfield S. N. (1992) Object oriented system analysis : a model driven approach. Yourdon Press

Flanagan D. (2011) JavaScript: The Definitive Guide, 6th ed. O'Reilly Media

Flanagan D., Matsumoto Y. (2008) The Ruby Programming Language, 1st ed. O'Reilly

Fowler M., Beck K., Brant J., Opdyke W., Roberts D., Gamma E. (1999) Refactoring: Improving the Design of Existing Code English, 1st ed. Addison-Wesley Professional

Frank U. (2000) Delegation: An Important Concept for the Appropriate Design of Object Models. In: Journal of Object-Oriented Programming 13(3), pp. 13–18

Frank U. (2014) Multilevel Modeling: Toward a New Paradigm of Conceptual Modeling and Information Systems Design. In: Business and Information Systems Engineering 6(6), pp. 319–337

Frank U. (2018) The Flexible Modelling and Execution Language (FMMLx) – Version 2.0: Analysis of Requirements and Technical Terminology

Frank U., Clark T. (2023) Language Engineering for Multi-Level Modeling (LE4MM): A Long-Term Project to Promote the Integrated Development of Languages, Models and Code. In: Proceedings of the Research Projects Exhibition at the 35th International Conference on Advanced Information Systems Engineering (CAiSE 2023). CEUR, pp. 97–104

Frank U., Mattei L. L., Clark T., Töpel D. (2022) Beyond Low Code Platforms: The XModelerML - an Integrated Multi-Level Modeling and Execution Environment. In: Proceedings of the Modellierung 2022 Satellite Events. GI, pp. 235–244

Gamma E., Helm R., Johnson R., Vlissides J. (1994) Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc.

Guarino N., Carrara M., Giaretta P. (1994) Formalizing Ontological Commitments. In: Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence. AAAI'94. AAAI Press, Seattle, Washington, pp. 560–567

Halpin T. (1995) Conceptual schema and relational database design, 2nd ed. Prentice Hall

Herrmann S. (2003) Object Teams: Improving Modularity for Crosscutting Collaborations. In: Aksit M., Mezini M., Unland R. (eds.) Objects, Components, Architectures, Services, and Applications for a Networked World. Springer, pp. 248–264

Jäkel T. (2017) Role-Based Data Management. PhD thesis, Technischen Universität Dresden

Kappel G., Schrefl M. (1991) Object/behavior diagrams. In: [1991] Proceedings. Seventh International Conference on Data Engineering, pp. 530–539

Kegel H., Steimann F. (May 2008) Systematically refactoring inheritance to delegation in java. In: 30th International Conference on Software Engineering (ICSE 2008), pp. 431–440

Kiczales G., Des Rivieres J., Bobrow D. G. (1991) The art of the metaobject protocol. MIT press

Knig D., King P., Laforge G., D'Arcy H., Champeau C., Pragt E., Skeet J. (2015) Groovy in Action, 2nd ed. Manning Publications Co.

Kühn T. (2017) A Family of Role Modeling Languages. PhD thesis, Technischen Universität Dresden

Kühn T., Leuthäuser M., Götz S., Seidl C., Aßmann U. (2014) A Metamodel Family for Role-Based Modeling and Programming Languages In: Software Language Engineering: 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings Springer, pp. 141–160

Leuthäuser M. (2015) SCROLL - A Scala-based library for Roles at Runtime.

Leuthäuser M., Aßmann U. (2015) Enabling View-based Programming with SCROLL: Using Roles and Dynamic Dispatch for Etablishing View-based Programming. In: Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering. MORSE/VAO '15. ACM, L'Aquila, Italy, pp. 25–33

Lieberman H. (1986) Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86). ACM, pp. 214–223

Liskov B. H., Wing J. M. (1994) A Behavioral Notion of Subtyping. In: ACM Transactions on Programming Languages and Systems 16, pp. 1811–1841

Matsakis N. D., Klock II F. S. (2014) The Rust Language. In: Ada Lett. 34(3), pp. 103–104

Neumayr B., Grün K., Schrefl M. (2009) Multi-level Domain Modeling with M-objects and M-relationships. In: Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling - Volume 96. APCCM '09. Australian Computer Society, Inc., Wellington, New Zealand, pp. 107–116

Neumayr B., Jeusfeld M. A., Schrefl M., Schütz C. (2014) Dual Deep Instantiation and Its ConceptBase Implementation. In: Advanced Information Systems Engineering. Springer, pp. 503–517

Odersky M., Spoon L., Venners B. (2016) Programming in Scala: Updated for Scala 2.12, 3rd ed. Artima Incorporation

Orrù M., Tempero E. D., Marchesi M., Tonelli R. (2015) How Do Python Programs Use Inheritance? A Replication Study. In: 2015 Asia-Pacific Software Engineering Conference, pp. 309–315

Pernici B. (1990) Objects with Roles. In: Proceedings of the ACM SIGOIS and IEEE CS TC-OA Conference on Office Information Systems. COCS '90. ACM, Cambridge, Massachusetts, USA, pp. 205–215

Prototype-Based Programming: Concepts, Languages and Applications. Springer

Raj R. K., Tempero E. D., Levy H. M., Black A. P., Hutchinson N. C., Jul E. (1991) Emerald: A General-Purpose Programming Language. In: Softw., Pract. Exper. 21(1), pp. 91–118

Rau K.-H. (2016) Agile objektorientierte Software-Entwicklung ger, 1. Aufl. 2016. Springer

Riehle D. (2000) Framework Design: A Role Modeling Approach. PhD thesis, ETH Zürich

Ruby Programming Language. Last Access: Accessed: February 2, 2024

Rumbaugh J. (1991) Object oriented modeling and design eng. Prentice-Hall Intl.

Schärli N., Ducasse S., Nierstrasz O., Black A. P. (2003) Traits: Composable units of behaviour. In: European Conference on Object-Oriented Programming. Springer, pp. 248–274

Selçuk Y. E., Erdoğan N. (2004) JAWIRO: Enhancing Java with Roles. In: Computer and Information Sciences - ISCIS 2004. Springer, pp. 927–934

Selçuk Y. E., Erdoğan N. (2006) A Role Model for Description of Agent Behavior and Coordination. In: Engineering Societies in the Agents World VI. Springer, pp. 29–48

Selic B., Gullekson G., Ward P. T. (1994) Real-time object-oriented modeling

Sowa J. (1988) Using a Lexicon of Canonical Graphs in a Semantic Interpreter: Relational Models of the Lexicon. In: Evens M. (ed.). Cambridge University Press, pp. 113–137

Stefik M., Bobrow D. (Jan. 1986) Object-oriented Programming: Themes and Variations. In: AI Mag. 6(4), pp. 40–62

Steimann F. (2000a) Formale Modellierung mit Rollen. Habilitationsschrift Universität Hannover

Steimann F. (2000b) On the Representation of Roles in Object-oriented and Conceptual Modelling. In: Data Knowl. Eng. 35, pp. 83–106

Steimann F. (2001) Role = Interface - A merger of concepts. In: Journal of Object-Oriented Programming 14, pp. 23–32

Steimann F. (2006) The Paradoxical Success of Aspect-oriented Programming. In: SIGPLAN Not. 41(10), pp. 481–497

Stein L. A. (1987) Delegation is Inheritance. In: Object-oriented programming systems, languages and applications (OOPSLA 1983). Conference Proceedings, pp. 138–146

Strahringer S. (1998) Ways of Handling and Interpreting Specialization in Object-Oriented Modeling In: The Unified Modeling Language: Technical Aspects and Applications Physica-Verlag HD, pp. 170–189

Szyperski C., Gruntz D., Murer S. (2002) Component software: Beyond object oriented programming, 2nd ed. The Addison Wesley component software series. Addison-Wesley

Taylor D. A. (1990) Object oriented technology: a manager's guide. Addison-Wesley

Tempero E. D., Yang H. Y., Noble J. (2013) What Programmers Do with Inheritance in Java. In: ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings, pp. 577–601

Turon A. (2015) Abstraction without overhead: Traits in Rust

Yourdon E. (1994) Object oriented systems design: an integrated appraoch eng. Yourdon

Zivkovic S., Karagiannis D. (2016) Mixins and Extenders for Modular Metamodel Customisation. In: 18th International Conference on Enterprise Information Systems

Appendix

Data structures used by the XModeler virtual machine

```
datatype Obj { of:Obj; slots:[Slot] }
datatype Slot { name:Str; value:Val }
datatype Cons(H,T) { head:H; tail:T }
datatype Nil { }
[T] = Nil + Cons(T,[T])
datatype {T} { elements:[T] }
Val = Obj + [Val] + {Val} + Fun + Atom
Atom = Str + Int + Float + Bool
Code = [Instr]
datatype Fun {
  arity:Int; code:Code; globals:[Val];
    dynamics:[Val];
  self:Obj; properties:[Slot]; supers:[Fun
    ]; sig:Obj }
```

Figure 20: XModeler VM Data Structures

Core syntax of the XOCL language

```
e ::=                          expressions
  | a                          simple exps
  | let i = e,... in e         locals
  | [e,...]                    lists
  | [ e | i <- e,...]           comprehensions
  | fun(n,...) e                abstractions
  | if e then e else e         tests
  | e(e,...)                   applications
  | k(e,...)                   kernel calls
  | e.i(e,...)                 messages
  | e.i                        slot access
  | e.i := e                   slot update
  | throw e                    exceptions
  | while e do e end           loops
  | case e of p → e,... end pattern matching
  | e;e                        ordering
a ::=                          atomic values
    i                          variable
  | i(::i)+                    lookup
  | s | b | n                  string, boolean, number
k ::=                          kernel calls
  | Kernel_of                  classifier
  | Kernel_setOf               set classifier
  | Kernel_getSlotValue        slot access
  | Kernel_hasSlot             slot availability
  | Kernel_setSlotValue        slot update
  | Kernel_mkSlot              slot creation
  | Kernel_mkObj               object creation
  | Kernel_invoke              fun-application
  | Kernel_addSlot             add a slot
```

Figure 21: XOCL Language

Essential XCore bootstrap definitions

```
mkClass(c,name,parents,atts) =
  Kernel_setOf(c,Class);
  Kernel_addSlot(c,"name",name);
  Kernel_addSlot(c,"parents",parents);
  Kernel_addSlot(c,"attributes",atts);
  c

mkAtt(name,type) =
  let a = Kernel_mkObj(Attribute,[])
  in Kernel_addSlot(a,"name",name);
     Kernel_addSlot(a,"type",type);

Class     = Kernel_mkObj(null,[])
Attribute = Kernel_mkObj(null,[])

mkClass(Attribute,"Attribute",
  [StructuralFeature,DocumentedElement],
  [mkAtt("name",String),mkAtt("type",
    Classifier)])

mkClass(Class,"Class",
  [Classifier],
  [mkAtt("attributes",mkSeqType(Attribute)
    )])
```

Figure 22: Excerpt of XCore bootstrap definitions

XOCL evaluation algorithm

```
eval(exp,env,o) =
  case exp of
    Var(n)          → env.lookup(n) when env.binds(n);
    Var(n)          → o.get(n);
    Path([n|ns])    → eval(Var(n),env,o).lookup(ns);
    Int(n)          → n;
    Bool(b)         → b;
    Str(s)          → s;
    Let(n,esub1,esub2)      → eval(esub2,env[n mapsto eval(esub1,env,o)],o)
    List(listofe)         → [ eval(e,env,o) | e <- listofe ];
    Cmp(e,[])       → [];
    Cmp(esub1,[n <- esub2]) →
      [ eval(esub1,env[n mapsto v],o) | v <- eval(esub2,env,o) ];
    Cmp(e,[b | listofb ]) →
       eval(Cmp(Cmp(e,listofb),[b]),env,o).flatten();
    Fun(listofn,e)      → InterpretedOperation(listofn,env,o,e);
    If(esub1,esub2,esub3)     → eval(esub2,env,o) when eval(esub1,env,o);
    If(esub1,esub2,esub3)     → eval(esub3,env,o);
    App(e,listofe)      →
      eval(e,env,o).apply([eval(e,env,o) | e <- listofe]);
    Send(e,n,listofe)   →
      eval(e,env,o).send(n,[eval(e,env,o) | e <- listofe]);
    Get(e,n)      → eval(e,env,o).get(n);
    Set(esub1,n,esub2)    → eval(esub1,env,o).set(n,eval(esub2,env,o));
    Order(esub1,esub2)    → eval(esub1,env,o); eval(esub2,env,o);
    Throw(e)      → Kernel_throw eval(e,env,o);
    While(esub1,esub2)    →
      eval(esub2,env,o); eval(exp,env,o) when eval(esub1,env,o);
    While(esub1,esub2)    → null
  end
```

Figure 23: Operational rules of the XOCL interpreter