The Flexible Multi-Level Modelling and Execution Language (FMMLx)

Frank, Ulrich

In: ICB Research Reports - Forschungsberichte des ICB / 2018

DOI: https://doi.org/10.17185/duepublico/47506

URN: urn:nbn:de:hbz:464-20181221-103049-3

Link: https://duepublico.uni-duisburg-essen.de:443/servlets/DocumentServlet?id=47506

Source: ICB-Research Report No. 66, December 2018

ICB
Institut für Informatik und Wirtschaftsinformatik

Ulrich Frank

ICB-RESEARCH REPORT 66

The Flexible Multi-Level Modelling and Execution Language (FMML^x)

Version 2.0: Analysis of Requirements and Technical Terminology

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

ICB-Research Report No. 66
December 2018

Authors' Address:

Ulrich Frank
University of Duisburg-Essen
Institute for Computer Science and Business Informatics
Universitätsstr. 9, 45141 Essen, Germany
Email: ulrich.frank@uni-due.de

Contact:
Institute for Computer Science and
Business Information Systems (ICB)
University of Duisburg-Essen
Universitätsstr. 9
45141 Essen – Germany
Tel.: +49 201-183-4041
Fax: +49 201-183-4011
Email: icb@uni-duisburg-essen.de

ISSN 1860-2770 (Print)
ISSN 1866-5101 (Online)
DOI 10.17185/duepublico/47506

Abstract

The Flexible Meta Modelling and Execution Language (FMML^x) is a multi-level language that allows to create executable models. It is implemented in the integrated meta-programming and meta-modelling environment Xmodeler. The Xmodeler is based on XCore, a recursive, reflexive meta-model that allows the definition and implementation of classes on arbitrary classification levels. However, XCore does not allow to assign classification levels to classes. The FMML^x extends XCore with explicit classification levels and intrinsic features that allow for deferred instantiation. Like XCore, the FMML^x features a common representation of models and code. As a consequence, models can be executed without the need to transform them to code. The use of the FMML^x during the last five years was promising, since it allows to clearly promote reuse and flexibility of modelling languages and software systems. Nevertheless, some of the assumptions its design was based on had to be challenged. In addition, the use of the language was compromised by various limitations. Against this background, it was decided to develop a substantially revised version, the FMML^x 2.0. This report presents an analysis of requirements that form the foundation for the subsequent specification of the new version. The report also presents a proposal for a technical terminology of multi-level modelling.

Keywords: Multi-Level Modelling, DSML, Language Architecture, Requirements

Contents

1 Introduction

Multi-level modelling is motivated by the shortcomings of traditional (modelling) language architectures that are usually restricted to one or two classification levels, and that allow for "shallow instantiation" only, which "is based on the premise that a class can only define the semantics of its direct instances, and can have no effect on entities created by further instantiation steps" (Atkinson and Kühne 2001, p. 20). In contrast to traditional language architectures, multi-level modelling languages enable an arbitrary number of classification levels. Also, every class is an object that may have a state and may respond to messages. Furthermore, they allow to define properties of a class that do not apply to their direct instances, but also to instances further down the instantiation chain. Hence, multi-level modelling enables a higher level of abstraction, which in turn promotes reuse and adaptability. It also enables to avoid the "accidential complexity" (Atkinson and Kühne 2008; Brooks 1995), caused by overloading traditional models. Multi-level modelling allows for creating models that include objects on multiple classification levels. Thus, modelers are no longer challenged by the mandatory decision between language and model layer, since each layer (apart from M_1) may serve as both, a representation of a language or of a model. In other words, a multi-level modelling environment empowers its users to be not just modelers, in the sense of applying a given modelling language, but to be language engineers at the same time. Hence, a multi-level modelling language qualifies as both, a meta-modelling language and a modelling language.

Various languages and tools have been proposed to support multi-level modelling (Atkinson and Gerbig 2016; Atkinson, Kennel, and Goss 2011; Jeusfeld 2009; Kühne and Schreiber 2007; Lara and Guerra 2010; Volz 2011). The basic concepts they use are similar, and in part equivalent. Differences often relate to terminology, and a few specific peculiarities. The latter include the question how multi-level models can be mapped to implementation languages. The Flexible Meta Modelling and Execution Language (FMML^x) is a multi-level language that allows to create executable models (Frank 2014). Different from other multi-level modelling languages, it features a common representation of conceptual models and code. This is accomplished by extending XCore, the meta-model of a meta-programming and modelling language that is supplemented with a comprehensive software development and (meta-) modelling environment, the Xmodeler (Clark, Sammut, and Willans 2008; Clark and Willans 2012). As a consequence, the notorious sychronization problem can largely be avoided. During the use of the FMML^x and the Xmodeler, various limitations were discovered that finally resulted in the decision to develop a new version of the language.

To prepare for the specification of the FMML^x 2.0, this report presents an analysis of requirements that go beyond the scope of the previous version. Multi-level modelling introduces concepts that do not only challenge established principles of conceptual modelling, but also demand for new perspectives on the modelling subject. Therefore, one can speak of a new paradigm. A new paradigm is usually characterized by the need for a new terminology. This is indeed the case for multi-level modelling. Since it introduces concepts that do not exist in the traditional paradigm, existing terms are not always suited to adequately express them. Therefore, the report also aims at clarifying the terminology in order to make descriptions of multi-level models more comprehensive and consistent.

2 New Requirements

The current version of the FMML^x has been in use for about five years, both for teaching purposes and within various research projects (Frank 2016; Kaczmarek-Heß 2017; Kaczmarek-Heß and Heß 2018; Kaczmarek-Heß and Kinderen 2017). Fig. 2.1 shows the meta-model of the current version. It extends XCore (Clark, Sammut, and Willans 2008) with additional properties to support intrinsic features and deferred (or "deep") instantiation. An intrinsic feature is an attribute, an operation, or an association that is not instantiated with the instantiation of a class on level m, but only on a predefined instantiation level n < m-1. The additional meta-attributes **`isIntrinsic`** and **`instLevel`** allow for the specification of intrinsic features. Furthermore, the classes **`MetaAdaptor`** and **`MetaClass`** were introduced to allow the direct creation of classes on any classification level.

The concrete syntax serves to clearly represent the peculiarities of multi-level models (see Fig. 2.2). The level of a class is indicated by the background colour of the area that serves the display of the class name. The name of the (meta-) class of a class is depicted on top of the class name. Intrinsic features are marked through the corresponding instantiation level, which is shown in white on a black square next to the representation of the feature.

The FMML^x was implemented in the Xmodeler, which enabled its use not only as a modelling language, but also as an implementation language, since models and code share the same representation. The language and the development environment demonstrated clear advantages over the traditional paradigm. At the same time, it became obvious that in some cases the presuppositions made with the design of the current version were not appropriate. Furthermore, the development and maintenance of larger models in the tool created the need for more elaborate support of model analysis and management. The following requirements are grouped into core language requirements (RC), requirements that result from the management and analysis of models with a modelling tool (RT), and requirements related to the notation of the language (RN). The priority of a requirement reflects the need for a corresponding feature based on the experience made with the prior version.

The requirements are illustrated with diagrams that are depicted in the notation of the prior version of the FMML^x . In cases, where the previous notation is not sufficient to represent the intended concept, a preliminary notation is used. Note that classification levels are represented in a style that is different from the one proposed with the UML. Instead of representing the classification level with an integer in regular script, such as M2, the integer is printed in

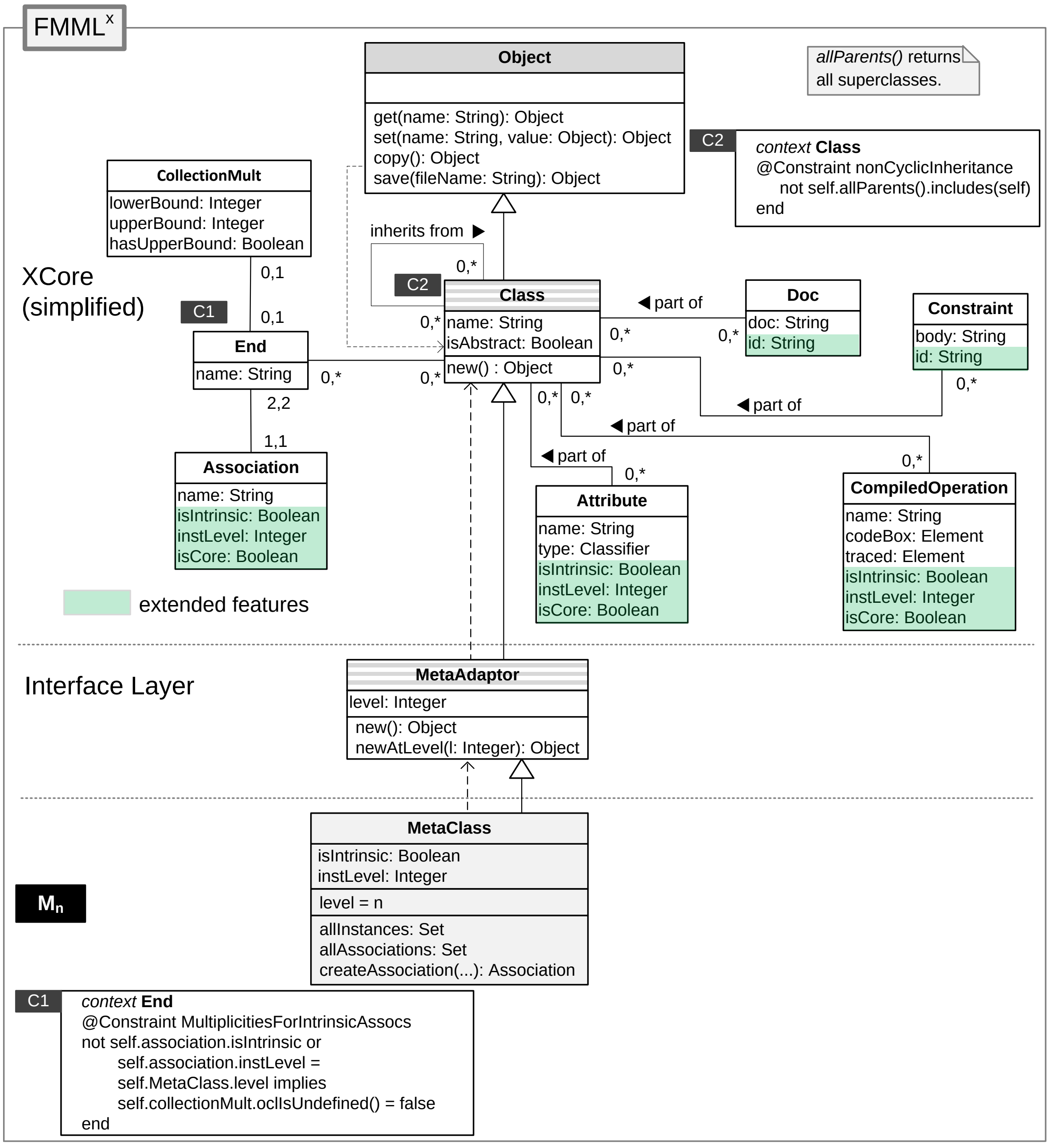

Figure 2.1: *Meta-Model of the Current Version*

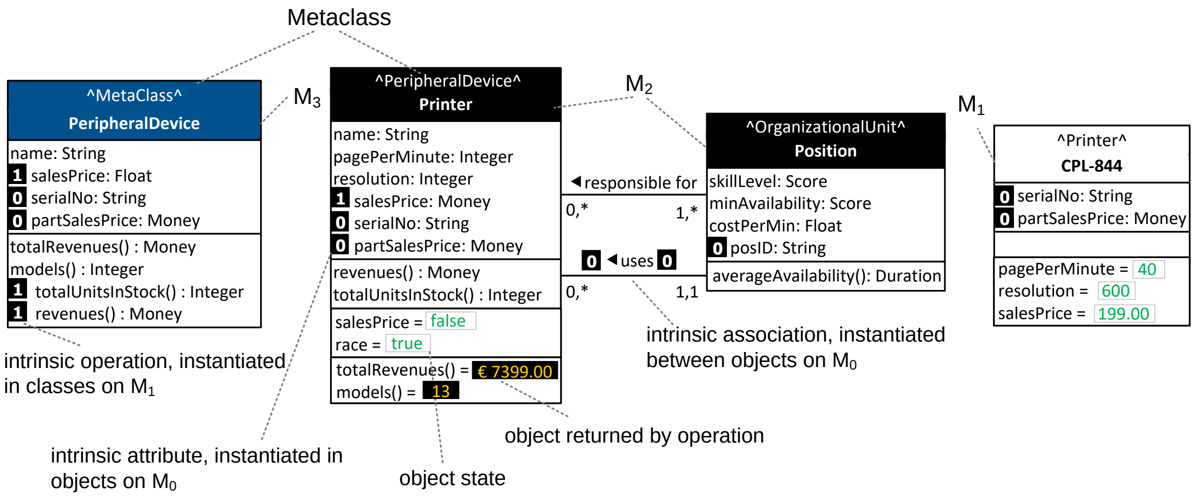

Figure 2.2: *Core Concepts of the Current Version and Corresponding Notation*

subscript. This is mainly for the reason that the semantics of levels in multi-level language architectures is different from that in traditional object-oriented language architectures like the one proposed by the MOF (*Meta Object Facility (MOF) Core Specification: Version 2.0* 2006).

2.1 Language Core

The core of a language comprises all concepts that are defined by its abstract syntax and semantics.

2.1.1 Contingent Levels

Every class specified in the current version of the FMML^x had to be assigned an explicit classification level. There is a good reason for this constraint, sometimes referred to as "strict multi-level modelling" (Atkinson and Kühne 2001, p. 28). With respect to the semantics of a class, it makes a clear difference whether it is meant as a class on M$_1$ or on any M$_n$ with $n > 1$. In natural language we can cope with the ambiguity of terms like "Product", which can be used to refer to a particular product, a type of product, or even to the set of all kinds of product types. In conceptual modelling it is preferable to avoid ambiguity. Nevertheless, there are cases where the need to assign a strict, invariant level to a class is problematic, because it may prevent expressing a useful abstraction, e.g., to increase reusability. Sometimes, the design of two classification hierarchies of similar objects results in a top level meta-class that makes sense for both hierarchies, but that would be located on a level different in one hierarchy from the one needed in the other hierarchy. That creates a situation that cannot be represented in the previous version of the FMML^x , because it requires to assign one and only one level to a class. The example in Fig. 2.3 illustrates this problem. In the example, the class **`Product`** on M_4 is instantiated into **`PeripheralDevice`** on M_3 and into **`Desk`** on M_2. While this kind of abstraction may be seen as useful from a modeler's perspective, it is not possible in the current version of the language. According to the experience gathered during the last years this is not an exotic exception, but occurs regularly.

The current version leaves three options to deal with this situation. First, one could aim at introducing further classes that make sense. In the example in Fig. 2.3, it is conceivable to insert a class like **`Furniture`** on M_3 to bridge the gap between **`Desk`** and **`Product`**. However, if these additional classes are not required for the targeted software system, they would increase complexity. Apart from that, it is not always possible to insert further classes that make sense. Second, one could introduce "dummy" levels to balance the number of levels in different hierarchies. However, this option is hardly satisfactory, since it would result in classes that have no specific meaning other than creating a confusing workaround. As a consequence, it would compromise the comprehensibility and maintainability of models. Third, one could allow for "jumping" levels. In that case, a class on level m would not only be instantiated in objects on the level m-1, but also in objects on level below m-1, which is confusing. Finally, one could create separate hierarchies. While this option will usually be preferable over dummy classes, it comes with an obvious drawback. If two hierarchies have clear commonalities, they would be neglected in the model, and, thus, compromise reuse.

RC1 In addition to having classes with a definite level, it should be possible to define classes with a *contingent* level. A contingent level allows for adapting the concrete level of a class to different contexts of use.

Rationale: It happens that two classes on different classification levels could be classified by the same meta-class. However, this can be achieved only, if the meta-class does not have a definite classification level. Instead, its level would have to be contingent with respect to the instantiation context.
Priority: high

There are cases, where the instantiation level of attributes is not clear at the level where an intrinsic attribute is defined. The class **`Product`** on M_4 shown in Fig. 2.4 includes, among others, the intrinsic attribute **`pricePerUnit`**. Often, the sales price is defined for a type of product, since every particular exemplar should be sold at the same price. In this case, the attribute should be instantiated on M_1. However, there are also cases where sales prices are individually defined for particular exemplars only, which would require an instantiation on M_0.

The need for intrinsic attributes with contingent instantiation levels becomes even more obvious with contingent level classes. The diagram in Fig. 2.5 depicts the contingent level class **`Product`**. If it is used in a software system for specialized dealership for computers

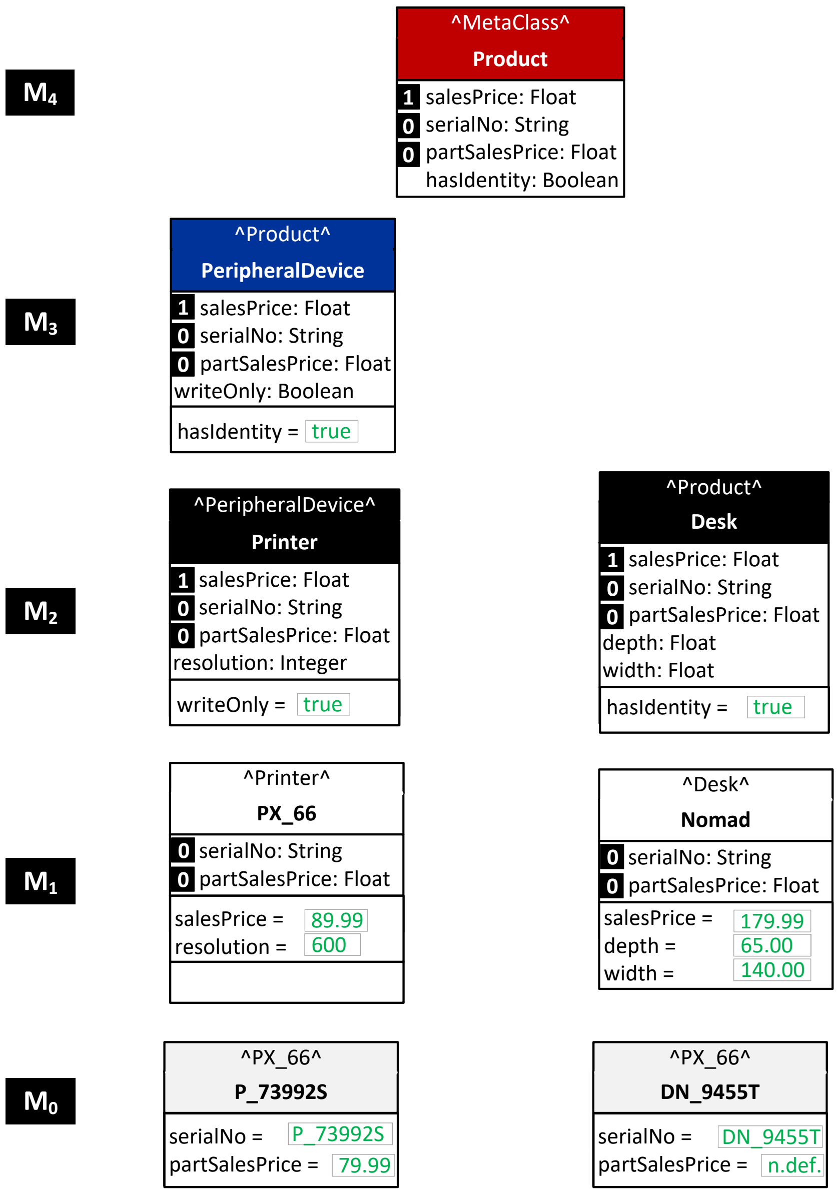

Figure 2.3: *Illustration of Problem Produced by Strict Levels*

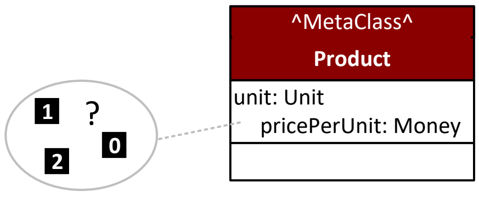

Figure 2.4: *Illustration of the Need for Contingent Instantiation Levels of Intrinsic Attributes*

and peripheral devices, it would be instantiated into classes representing peripheral devices or computers. If the class **`Product`** is used in a software system for a general store, it might be more appropriate to instantiate it directly into a class on M_1 that represents a particular product type. In the first case, the intrinsic attribute **`qualityLevel`** would be instantiated on M_2, in the second case, if would have to be instantiated on M_1.

RC2 It should be possible to define intrinsic attributes with a *contingent* instantiation level.

Rationale: It may not be possible to foresee the concrete instantiation level of an intrinsic attribute, while knowing that it has to be instantiated somewhere down the instantiation line.
Priority: medium

In addition to attributes, it is conceivable that intrinsic associations and intrinsic operations are also defined with contingent instantiation levels.

RC3 It should be possible to define intrinsic associations and operations with a *contingent* instantiation level.

Rationale: As with intrinsic attributes, it may not be possible to foresee the concrete instantiation level.
Priority: medium

Challenge: The definition of a class as level-contingent has a substantial impact on a model's integrity. It is not necessarily the case that an attribute defined in a contingent class can be instantiated on any level. If, for example, an attribute is intended to store the average customer satisfaction with a certain bicycle model or bicycle type, it would make sense only to instantiate it above M_1. Therefore, it may be required to add a minimum instantiation level. Similarly, the instantiation level of an intrinsic property can be defined on a level higher than the level of a class directly instantiated from a contingent class. For example, it would not be possible to instantiate the intrinsic operation **`averagePrice`** on the intended level 2. At the same time, it would not make sense on M_1.

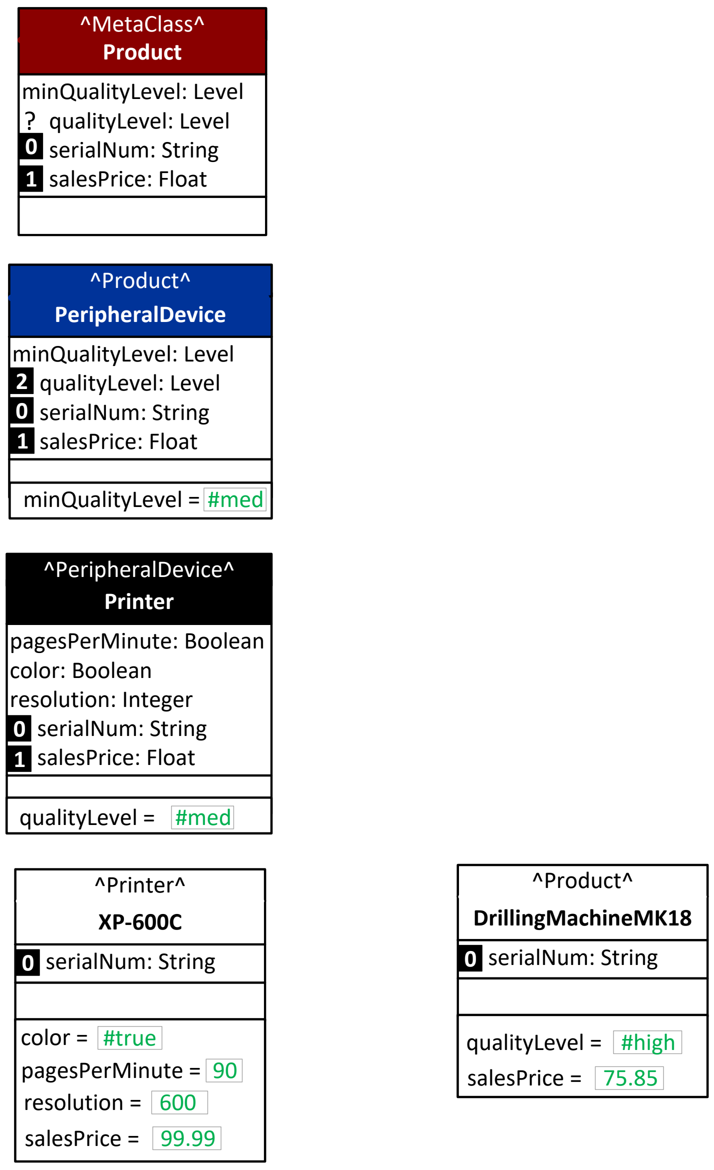

Figure 2.5: *Contingent Instantiation Levels of Intrinsic Attributes in Case of Contingent Level Classes*

RC4 It should be possible to define constraints on the intended instantiation level. These could address a minimum or maximum number of instantiation steps, or a set of alternative instantiation levels, e.g., level 1 or level 2.

Rationale: It is conceivable that at the time of specifying a class on level n, there is not sufficient knowledge available to exactly define an intended instantiation level. But usually at least some restrictions on possible instantiation levels will be known. Hence, the requirements follows from the principle that all knowledge available on a certain level should be specified there.
Priority: medium

Including constraints on intended instantiation levels into the language does not only add more concepts, but will also increase the complexity of the concrete syntax. Therefore, it seems appropriate to first do with an explicit specification of these constraints with the XOCL. If it turns out that these constraints are required often enough, they can still be built into the language.

Challenge: All three requirements demand for a cross-level constraint language, which will be available only with the new version of the XOCL.

2.1.2 "Classless" Classes

Like with any object-oriented models, the design of multi-level models requires the creation of classes. In traditional, one-level models, a class is implicitly instantiated from one specific meta-class. In multi-level models, a new class is instantiated from a class that exists in the model already. As a consequence, the design of a multi-level model requires a top-down approach, that is, one has to start with classes on the topmost classification level. Then, classes on lower levels have to be added step by step. There are cases where it may be perceived as inappropriate to be forced to a top-down approach. Sometimes, the topmost classes are not known in advance. Instead, they may result through an act of abstraction from lower level classes. In other words, it would be helpful, if a top-down approach was supplemented by a bottom-up approach. That would require allowing for the preliminary creation of classes without an explicit meta-class.

RC5 It should be possible to define preliminary classes without a meta-class.

Rationale: Sometimes, a bottom-up approach seems to be more appropriate than a top-down approach.
Priority: high

Challenge: From a technical perspective, it is not possible that a class does not have a meta-class. Therefore, some preliminary meta-class has to be assigned, which would be later replaced by the final meta-class. Hence, a corresponding modelling tool needs to support meta-class migration.

2.1.3 Distinction of Instantiation Levels within Associations

Like attributes and operations, associations could be defined as intrinsic, too. The example in Fig. 2.2 illustrates the idea. The classes **`Printer`** and **`Position`** are both located on M_2, that is, they are supposed to be instantiated into process types and position types. The intrinsic association between both is to express that a particular process instance on M_0 is assigned to a particular instance of a position type, also on M_0. This is indicated by the zero printed in white on the black square next to the association name. The specification of intrinsic associations in the meta-model of the current version (Fig. 2.1) was based on the assumption that the instantiation level of an intrinsic association has to be the same with both associated classes. It turned out, however, that this assumption is not appropriate. It is not a rare exception that an association should be instantiated on different levels with both associated classes. The example in Fig. 2.2 includes the association "responsible for". It is to express that a particular position instance on M_0 is responsible for one to many types of printers on M_1.

RC6 Intrinsic associations should allow the specification of different instantiation levels for both participating classes.

Rationale: It is common in natural language to link two concepts (or object references) on different classification levels in one sentence. Accordingly, it can be a relevant requirement to link two objects on different classification levels in an multi-level model. If the association can be defined on a higher level already, that is, if the association is intrinsic, it follows directly that it must be possible to support different instantiation levels.
Priority: high

This feature is already part of the current implementation of the FMML^x in the Xmodeler.

2.1.4 Deferred Specification of Associations

Intrinsic associations enable expressing knowledge about classes on a higher level of classification. We know, for instance, that some kind of shock absorbers are mounted to every car. This could be expressed as in the example in Fig. 2.7 on M_3. The association **`mounted_on`** is to be instantiated on M_1, that is, it allows the definition of the shock absorber types that can be mounted on a certain car model. However, at a higher level of classification one will usually not know the particular class(es) that are part of an intrinsic association at the level where it is

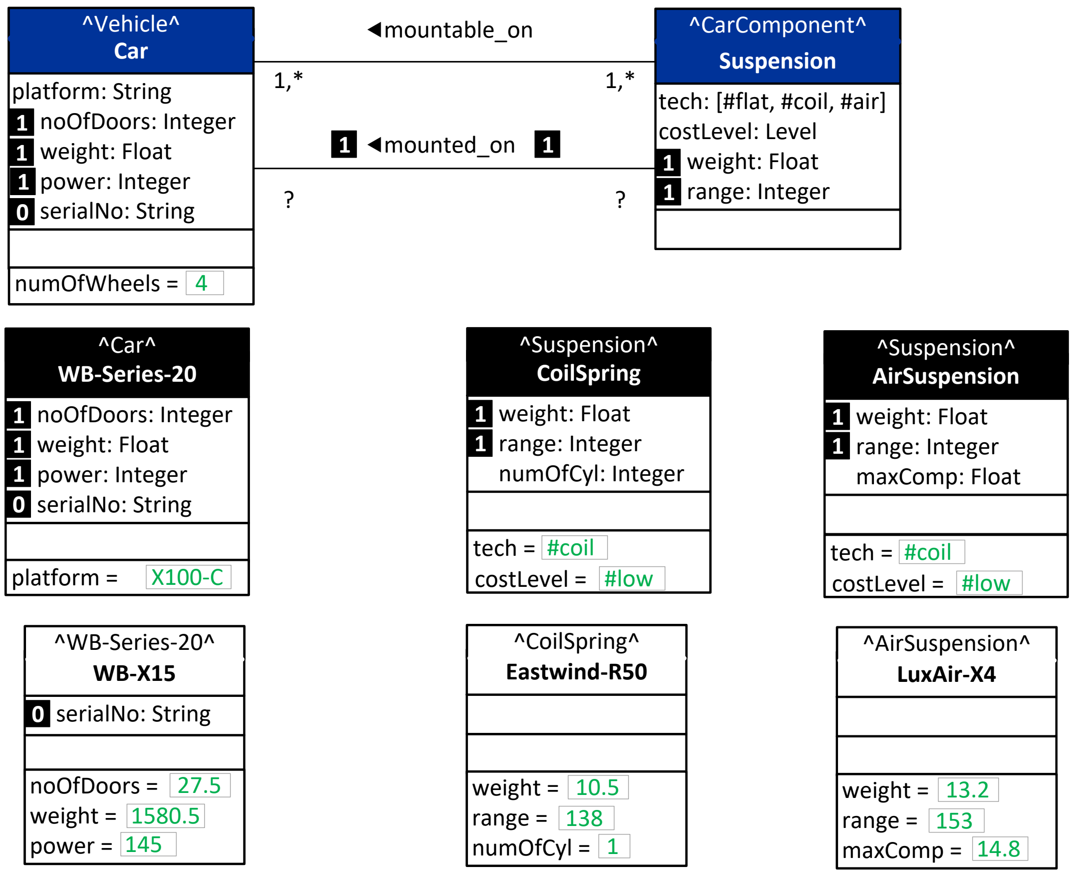

Figure 2.6: *Deferred Specification of Multiplicities*

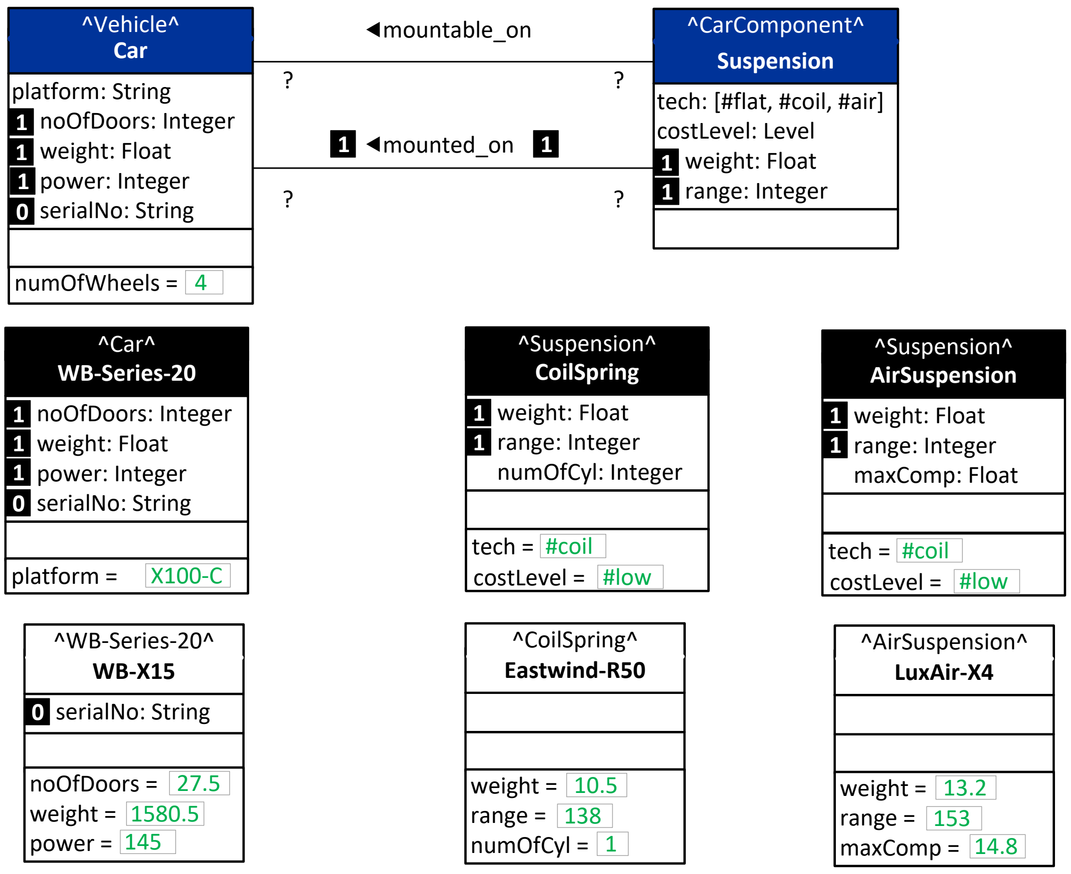

Figure 2.7: *Different Instantiation Levels of an Intrinsic Association*

to be instantiated. The example model in Fig. 2.7 would allow for any kind of car model being associated with any kind of shock absorber. However, it may be a strict rule that, e.g., a certain car series (in the example modelled on M_2) is produced with air suspension only. That would reduce the set of possible classes to form associations with classes that represent models of that car series to instances of **`AirSuspension`**. Furthermore, it may be characteristic of a car series that it does not allow for more than two types of shock absorbers. Since these restrictions may vary from car series to car series, they cannot be expressed with the general definition of the intrinsic association. As far as the possible classes on the level above the intended instantiation is concerned, there needs to be some kind of dynamically growing constraint. With every instantiation step the range of possible classes (and the respective instances and instances thereof) is reduced. This additional knowledge should be accounted for with the subsequent instantiation steps. The multiplicities of an intrinsic association may also not be predictable. A certain car series, for example, may be characterized by the rule to not allow for more than two types of air suspension. In that case, the specification of a particular multiplicity at the level where the intrinsic association is defined, would be misleading. To cope with such a situation, it would be useful to explicitly defer the specification of multiplicities of intrinsic associations.

RC7 The definition of intrinsic associations should imply that corresponding constraints are refined with every instantiation step.

Rationale: By their very nature, intrinsic associations lack certain knowledge about the actual association at the level where they are defined for. At the same time, with every instantiation step, the range of possible options decreases. Hence, refining corresponding constraints dynamically would be an important contribution to system integrity.
Priority: high

RC8 The language specification should allow for defining intrinsic associations without specific multiplicities. In other words, it should be possible to defer the specification of multiplicities to a lower level.

Rationale: In cases where the multiplicities of an intrinsic association at the intended association level may vary, the definition of a particular multiplicity would create ambiguity or even a contradiction, hence, a serious threat to a system's integrity.
Priority: high

The following requirement describes a special case of the previous one.

RC9 If absolute minimum or maximum cardinalities are known, they should serve as constraints for the final definition of multiplicities on the intended instantiation level. That means that the actual cardinalities specified on the intended instantiation level have to be within this range. As a consequence, it needs to be possible to mark cardinalities as boundary values.

Rationale: It should be possible to specify all knowledge about multiplicities of associations that is available.
Priority: high

Challenge: All three requirements can be satisfied only with a cross-level constraint language, which will be available only with the new version of the XOCL.

2.1.5 Avoiding Redundant Specification

It happens that classes within a classification hierarchy share the same properties, that is, the same attributes, operations, or associations. A frequent example is an attribute like "name", which may be required for a class, its instances and all further instances down the instantiation line. Further examples comprise operations that perform statistics on object populations. It might be required for every class within a classification hierarchy to provide methods that calculate the number of direct instances or the cumulated number of all instances of instances of a class (see example in Fig. 2.8). Normally, this would be a clear case of inheritance. In

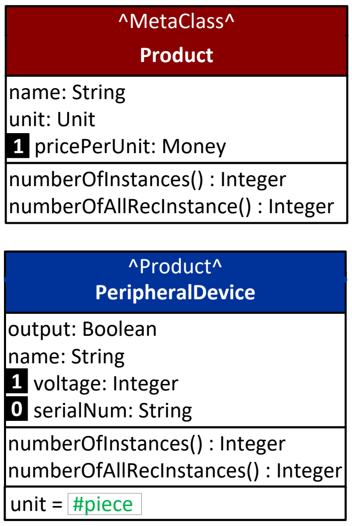

Figure 2.8: *Example of Repeated Specification of the Same Properties*

the current version, it was, e.g., not possible to indicate that a property of a class should be inherited instead of being instantiated (either directly or deferred in case of an intrinsic attribute). As a consequence, those features that are also required for instances of classes have to be duplicated, leading to the well-known problems caused by redundancy.

RC10 It should be possible to indicate that certain properties of a class are to be inherited to its instances.

Rationale: Inheriting features where it is appropriate enables preventing redundancy. Note that inheriting properties is different from intrinsic properties. An intrinsic property is not directly instantiated where it is defined, but only at the specified instantiation level. An inherited property can be part of many classes. Therefore, it can be instantiated multiple times. In an ideal case, the implementation of operations can be inherited, too (as it would be the case with the example in Fig. 2.8).

Priority: high

In the current implementation, this problem is relaxed implicitly. Certain properties, such as the attribute **`name`**, are defined with the central class in Xcore **`Class`**. Every class that is defined with the FMML^x inherits through **`MetaClass`** from **`Class`** (see Fig. 2.1). However, this approach works only for properties that are generic enough to be inherited to all classes of an entire system.

RC11 Intrinsic properties must not be explicitly inherited, since they are passed to instances anyway until the intended instantiation level is reached.

Rationale: Explicit inheritance of intrinsic properties would be redundant and would therefore compromise the comprehensibility of a model.

Name	*Class*	*Problem*
price	Float	a price could be negative or be unreasonable large
unit	String	any string could be used, typos would result in new units; knowledge about converting units could not be incorporated
currency	String	any string could be used, typos would result in new currencies; additional knowledge such as abbreviations, conversion of currencies, etc. would not be included.
level	Integer	depending on the scale, the attribute may allow for a few values only, e.g. 1 - 10; however, that cannot be expressed
multiplicity	Integer	would not allow for representing a symbol for unlimited such as "*"
multiplicity	Char	would allow for representing a symbol for unlimited, however, at the price to give up on arithmetics which are required for checking upper bound against lower bound.
color	String	would allow for any string and would not allow for including color-specific information such as a specific color model.

Table 2.1: *Problems Caused by Inappropriate Semantics of Attribute Classes*

Priority: high

2.1.6 Auxiliary (Meta-) Classes

Attributes are to a large extent specified with a small set of generic data types or classes. It would be possible to define more specific classes for this purpose. However, many modellers would regard that as too much effort. As a consequence, attributes remain underspecified in most cases. This will usually create a threat to system integrity. The examples in Table 2.1 illustrate this problem.

RC12 The language should be extended with auxiliary classes that represent specific concepts that are often needed. They include various units of measurement, currencies, multiplicities, and, more specific, economic concepts such as price.

Rationale: The availability of thoroughly defined auxiliary classes contributes to model integrity and modelling productivity.
Priority: high

RC13 The definition of specific auxiliary classes should be supported. That includes the definition of classes which represent enumerations or ranges of elements of a certain type/class.

Rationale: This is an incentive for software developers to define specific auxiliary classes and, thus, to improve model and system integrity.

Priority: high

The current implementation of the FMML^x includes already various auxiliary classes, but no support for the definition of specific auxiliary classes.

2.1.7 Differentiation of Uni- and Bi-Directional Associations

From a conceptual perspective, there is no need to qualify an association as uni- or bi-directional. However, as soon as a model is to be executed or mapped to code, a corresponding decision has to be made.

RC14 The specification of an association should allow for defining whether the association is bi-directional or uni-directional.

Rationale: Even though this information is required for implementation only, it should be possible to add it to a conceptual model to prepare for a smooth transition to design and implementation. This is even more the case for an executable modelling language like the FMML^x , since there is no clear borderline between models and code anymore.
Priority: high

2.1.8 Support of Delegation

Delegation is an important concept to promote reuse and flexibility (Szyperski, Gruntz, and Murer 2011). Its use in object-oriented languages is mainly motivated by counter-intuitive effects of inheritance, which create a threat to system integrity. These effects are caused by the fact that, different from logic and natural language, every object is of one and one class only. Fig. 2.9 illustrates the problem caused by an inappropriate, yet intuitively correct use of inheritance.

Delegation is an association between two classes, where the instances of one class serve as delegators, the instances of the other class as delegatee objects. A delegator object "inherits" behaviour and state from the corresponding delegatee object. The example in Fig. 2.10 illustrates the principle idea of delegation. Whenever an object of the class **`Student`** receives a message it cannot respond to, the message is delegated to the corresponding delegatee object of the **`Person`**. The object that is returned by the method in the object of the class **`Person`** is returned to the object that initially sent the message to the delegator object. Therefore, the interface of a delegator object is not only extended by the interface of a corresponding delegatee object, but it also has access to the delegator's state. Hence, delegation allows to mimic the semantics of specialization in logic, where an object of class B, which is specialized from class A, is an instance not only of B, but also of A. Delegation is not only useful to connect objects on M_0, but

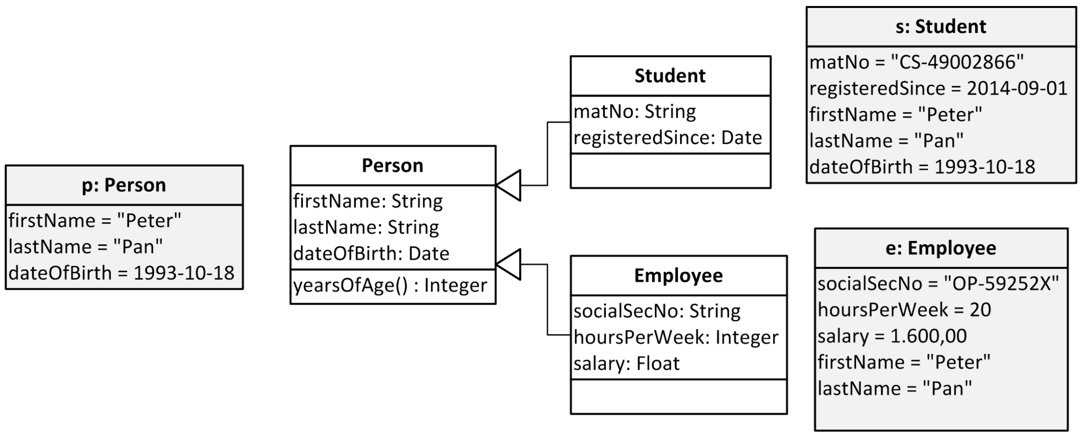

Figure 2.9: *Counter-Intuitive Effects of Inheritance*

can be applied to classes on any classification level, too. A common use case for connecting classes is the definition of a variant as a delegator of a core artefact (see Fig. 2.11)

RC15 Delegation should be enabled between classes on any classification level as a specific association between two classes on any level.

Rationale: In object-oriented system, delegation is an important abstraction that allows to compensate for misconceptions of inheritance.
Note that further, more specific, requirements would be needed for a comprehensive specification of delegation. An elaborate analysis of delegation is subject of a further publication that is currently under review. Apart from that, a preliminary implementation of delegation exists already in the current version of the language implementation.
Priority: high

2.1.9 Support for the Specification of Further Association Types

Associations are of pivotal relevance for the design of languages and models. Currently, association is not a meta-concept. The concept of an association is used, but not defined by XCore. Hence, it does not allow for the instantiation of more specific association types. As a consequence, there is only one generic kind of association provided with the current version of the FMML^x . If an association is characterized by specific semantics, it has to be expressed by additional constraints. It would be a clear improvement, if the language enabled the definition of specific association types. These could be more general types like aggregation, composition, and delegation. Note that delegation has been addressed by a dedicated requirements, because of its outstanding relevance. Table 2.2 gives an overview of general and association types.

In addition to general association types, there are domain-specific association types. Their semantics depends on domain-specific requirements, which may vary. Hence, the language

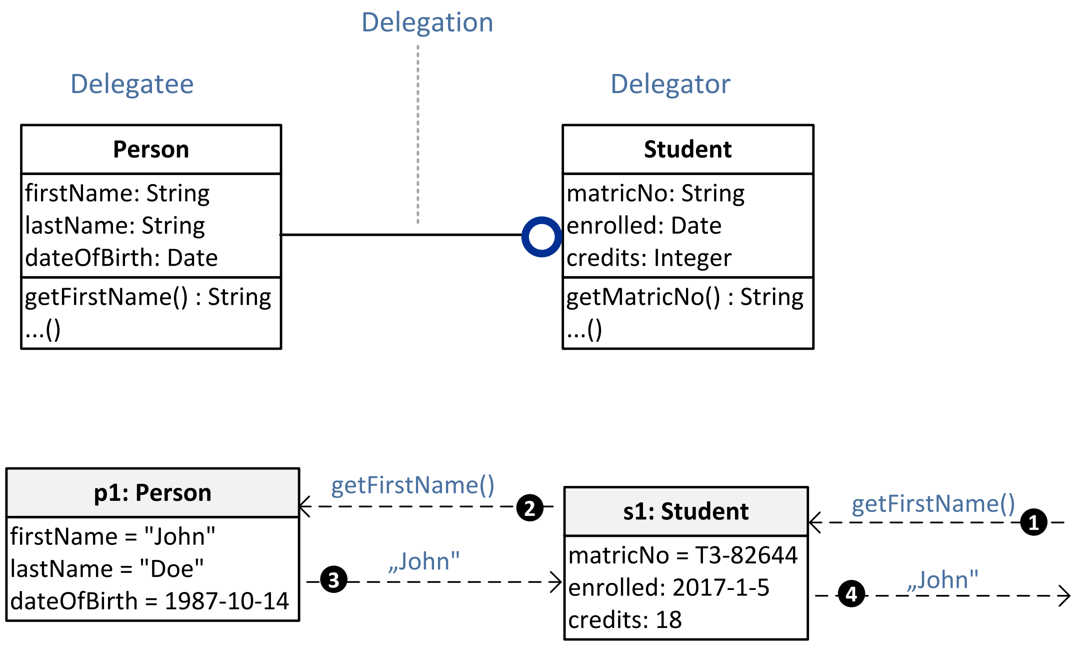

Figure 2.10: *Illustration of Delegation*

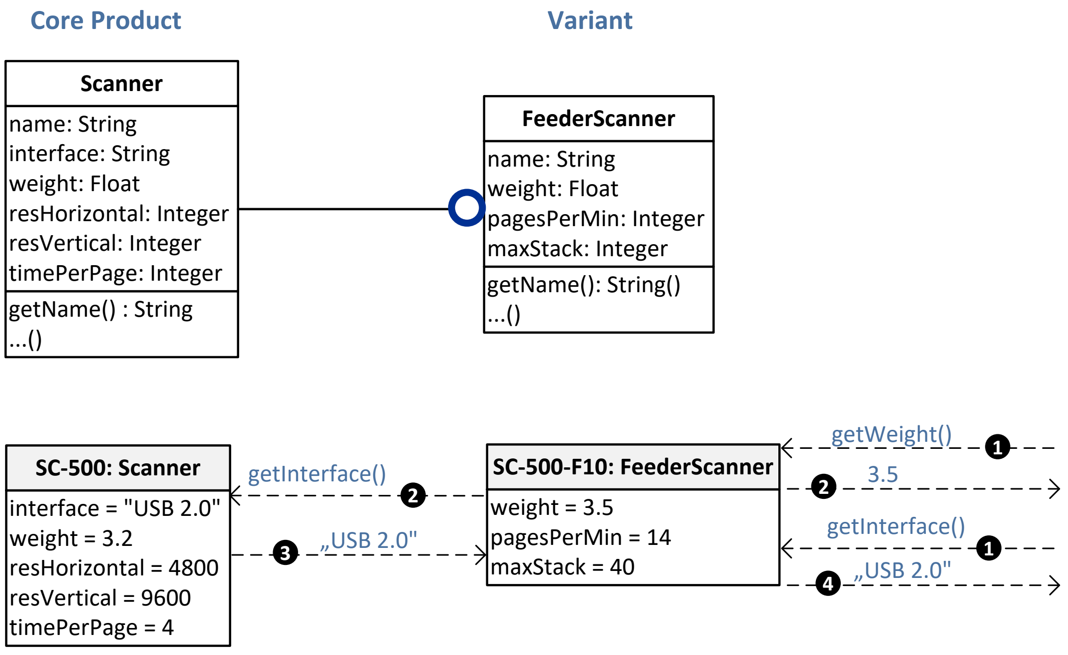

Figure 2.11: *Different Instantiation Levels of an Intrinsic Association*

Name	*Description*	*Semantics*	*Variants*	*Behaviour*
interaction	no specific semantics	none, restricted to cardinalities	state dependency, e.g., marriage, sex	transparent creation of links
aggregation	particular objects that are part of another object, e.g., a wheel that is part of a particular car	special case of interaction	state dependency; dependent on corresponding composition, e.g., restricted to instances of a certain class	transparent creation of links
composition	an abstraction over aggregation, e.g., the construction of a car model may allow for 1 .. * wheel types	similar to aggregation, however, on a different level of abstraction (a type is an aggregation of other types)	state dependency, e.g., any type of wheel that is certified for a certain speed	transparent creation of links including ownership
delegation	transparently access data or behaviour of another object	special case of interaction; *delegator* delegates behaviour (and, hence, state) to *delegatee*	state dependency, e.g., age, qualification, authorization	transparent creation of links; object creation; message dispatch
delegation to class	transparently access data or behaviour that is common to all instances of a class, e.g., a price of a product exemplar that is specified with the corresponding class	similar to delegation, however on a different level of abstraction. *Delegator* (object) delegates behaviour to its *class*	delegation may be restricted to classes that are marked as delegatees	transparent creation of links; object creation; message dispatch; requires rules to define dispatch order
documentation	linking to an object that needs to be documented at the time the link was created	special case of interaction (or aggregation), the linked object must not be changed during the lifetime of the association, e.g., a product linked to an invoice	partial documentation only, e.g., in an invoice, documentation could be restricted to products	as soon as the final state of an object that is to be documented, is committed, a copy needs to be created and set to read-only mode.

Table 2.2: *Example Association Types*

Name	*Kinds of classes*	*Description*	*Variations*
married to	Person, Person	marriage depends on certain requirements, e.g., sex, age	different constraints on sex, multiplicities, age
holds	Person, Driving Licence	holding a driving licence requires people to satisfy certain requirements	constraints on min. and max. age, gender, different types of driving licences
reminder to	Reminder, Customer	depends on the existence of a valid debt claim, term of payment overdue	term of payment, different kinds of debt claim, reminder charges, dunning level

Table 2.3: *Examples of Domain-Specific Association Types*

should offer meta-association types that allow the creation of various association types of the same kind. Also, the semantics of a domain-specific association type may depend on properties of the associated classes. However, the names of these classes and the corresponding properties may not be known in advance. Therefore, a language for the definition of specific association types should allow for using these names for the instantiation of an association type. Table 2.3 illustrates the idea of domain-specific association types with a few examples.

RC16 The language should provide generic association types. At best, these types would be instantiated from an association meta-type.

Rationale: Including generic association types promotes modelling productivity and model integrity. Meta-association types provide modellers with more flexibility, since they allow to vary the semantics of generic association types.
Priority: medium

RC17 It should be possible to define domain-specific association types. In order to cover the variance of domain-specific association types, it could be helpful, if a language provides meta-association types on a higher level, that would allow the instantiation of meta-association types.

Rationale: The specification of domain-specific association types promotes model integrity.
Priority: medium

Challenge: The specification of meta-associations is very demanding, because association types can hardly be defined on their own, that is, without regard to the classes they connect. For meta-associations, that would require adequate meta-classes, which, however, can hardly be predicted in advance.

2.1.10 Multiplicities of Attributes

There are cases where an attribute may be represented by more than one object of the attribute's class. For example, a person may have more than one telephone number, a bicycle frame could be painted with three different colours, etc.

RC18 It should be possible to specify multiplicities of attributes.

Rationale: Without this feature, it is hardly possible to adequately model cases where attributes are characterized by more than one object of a certain class.
There are not too many cases, where this requirement is relevant.
Priority: medium

2.1.11 Deferred Instantiation of Attribute Classes

In the current version, an attribute needs to be specified by a class on M_1. In most cases, this proved to be sufficient. However, with the specification of higher level classes one may know the existence of an intrinsic attribute without knowing its specific class. Instead, one may know a meta-class the attribute's class is instantiated from directly or indirectly, which corresponds directly to the instantiation of classes involved in an intrinsic association. Fig. 2.12 shows an example: we know that a bicycle frame is made of a certain material. Let us assume that **`BicycleFrame`** is located on M_3. It is then instantiated into different kinds of frames, such as racing frame or mountain bike frame, which are subsequently instantiated into certain types of frames. A bicycle frame in general is made of some kind of material, which could be expressed by the class **`Material`** on M_3. This meta class could then be instantiated into different kinds of material, such as metal or synthetics, which could be further instantiated into, e.g., steel, aluminum or carbon. The example in Fig. 2.12 shows the instantiation down to a certain alloy of aluminum on M_0. Note that one could also argue for placing it on M_1, because it rather represents a sort or type than a particular object. However, since it cannot be instantiated further, one might, for pragmatic reasons, decide placing it on M_0.

RC19 It should be possible to specify intrinsic attributes through classes on levels higher than M_1 together with an intended instantiation level of those classes.

Rationale: There are cases where it is not possible to clearly specify the class an intrinsic attribute will have at the level where it is supposed to be instantiated.
Due to our experience with the current version of the FMML^x the lack of deep instantiation of attributes was not a major shortcoming.
Priority: low

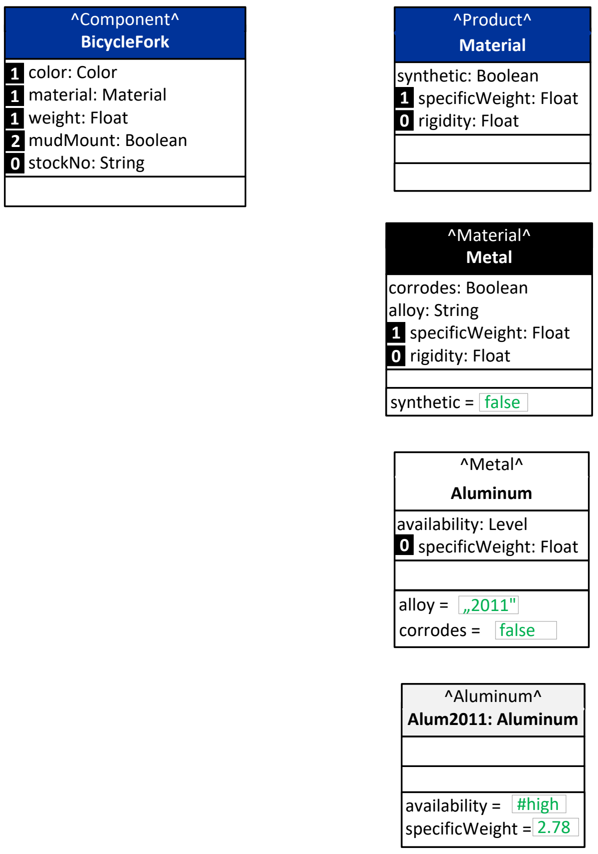

Figure 2.12: *Illustration of Deep Instantiation of Attributes*

2.1.12 Specification of Dependencies between Model Elements

The more semantics included in a model, the better are the chances to have it properly interpreted by a machine and to modify without compromising its integrity. Sometimes, the analysis of model elements requires additional information, which cannot be expressed by a modelling language.

Properties of a class in a multi-level system may serve different purposes. An attribute that is instantiated in a class may represent properties of that class, a property value shared by all instances of the class, or constraints on possible property values within instances. The example in Fig. 2.13 illustrates this issue with respect to attributes. The attribute **`minQualityLevel`** is instantiated with all instances and serves the specification of a constraint on possible values of instances of these instances (all types of printers in that system). The attribute **`serialNum`**, though being an intrinsic attribute, is supposed to be instantiated in individual values for each instance (particular printers). Finally, attributes like **`pagesPerMin`** are to be instantiated into values that are shared by all instances of the class a particular value was assigned to.

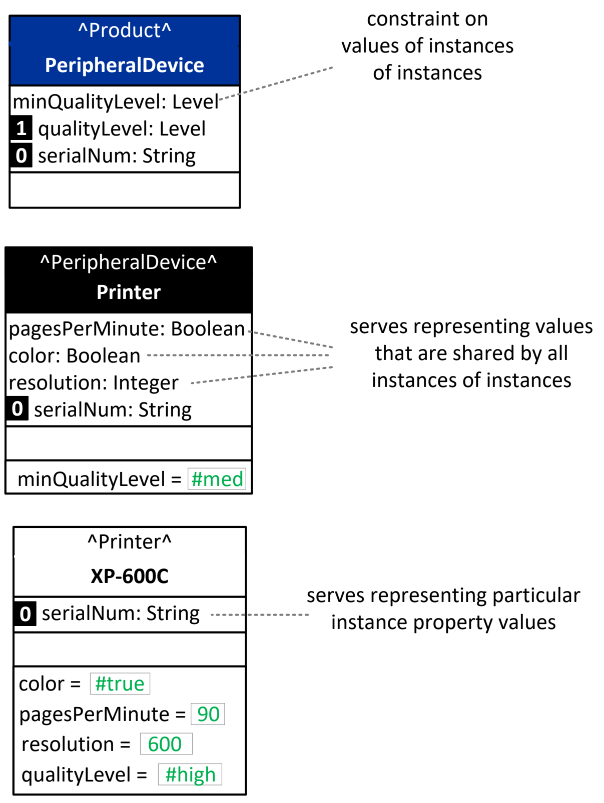

Figure 2.13: *Different Kinds of Attributes*

Similar distinctions can be made for operations. The values delivered or used by access operations would represent corresponding aspects. Associations may also be used to create links between an object (e.g., representing a country) and a class (e.g., representing a specific product type), where the link is semantically shared by all its instances.

RC20 The new version should allow for clearly distinguishing between attributes that are regularly instantiated into individual values of instances, that are instantiated into values of instances that actually represent common values of their instances, and those that serve the specification of constraints on attribute values. At the same time, corresponding access operations should be qualified accordingly.

Rationale: These different kind of properties have specific semantics that would be lost, if there was no way to express it somehow, which in turn could compromise the integrity of systems.
This requirement is fairly easy to satisfy and promises clear benefits.
Priority: high

There are also different kinds of operations. Generic categories of operations include *instantiation*, *read access*, *write access*, *update* or *release*. Furthermore, there are domain-specific categories of operations, e.g., *debit*, *booking*, *planning*, etc. If an operation is supplemented with information about the categories it belongs to, more meaningful analysis is enabled. Furthermore, it can contribute to model integrity. If, e.g., all booking operations of a ERP system need to be updated due to new requirements, a corresponding category would make sure to find all affected operations. This would foster maintainability, and as a consequence, system integrity.

RC21 It should be possible to assign categories to operations.

Rationale: Distinguishing categories of operations enables more meaningful model queries and fosters models integrity.
Priority: high

Like most modelling languages, the FMML^x does not allow for expressing certain dependencies between model elements. Access operations are particularly challenging with respect to model changes. The name of an access operation will usually reflect the name of the attribute it refers to, or the name of an associated class, the objects of which are accessed by an operation. If attributes or associations are deleted or modified, the corresponding access attributes need to be deleted or modified, too. A common approach to find the affected operations depends on naming conventions. While naming conventions are useful to foster the readability of a model, depending on them for finding affected operations is not entirely satisfactory. First, it would not allow to use names for operations other than those defined by the naming conventions. Second, it would require searching for a certain name pattern in all operations of a class. If the representation of an operation is augmented by the specification of the attribute or associated class it allows to access (in addition to the type of access), finding affected operations would be clearly more convenient.

RC22 It should be possible to link access operations to the objects they refer to, such as attributes or associations.

Rationale: This additional information would facilitate the updating of operations after relevant aspects of the reference object had been changed.
Priority: high

Dependencies may also exist between associations, which recommends making them explicit, too.

RC23 The new version should support the specification of an association as dependant of another one. That is, the instantiation of the dependant association must not happen without the association it depends on being instantiated, too.

Rationale: Including a corresponding concept in the language does not only free modellers from the specification of a corresponding constraint, it also increases the likelihood that the dependency is explicitly modelled at all.

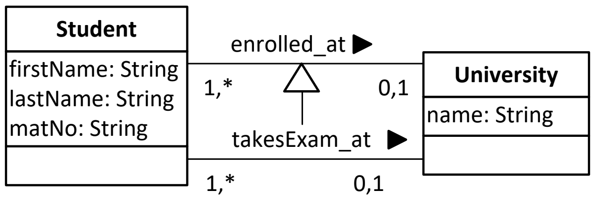

Figure 2.14: *Illustration of Association Specialization with the UML*

Priority: medium

In addition generic dependencies, there may be more specific kinds of dependencies between associations. A known example of this kind is so called association specialization in the UML. If an association A1 is specialized from another association A0 between the classes C1 and C2, then the A1 connects C1 and C2, too, or subclasses of the two (OMG 2010, p. 112). The idea is that the set of links instantiated from the specialized association is a subset of the links instantiated from the generalized association. Fig. 2.14 illustrates this idea.

However, this would not be the case for specialized associations that connect subclasses of the classes connected by the generalized association, since in object-oriented systems where an object is of one and only one class, an instance of a subclass cannot be an instance of a superclass at the same time. Therefore, the UML specification of association specialization is not satisfactory. Instead of allowing for subclasses in the specialized association, delegation would be a better choice (see Fig. 2.15). The class **`StudentAssistant`** defines roles (delegators) of the role filler (delegatee) class **`Student`**. Every instance of **`StudentAssistant`** is linked to exactly one corresponding instance of **`Student`**, and has transparent access to its state. Therefore, the instances of a role class can be seen as a subset of the instances of the corresponding role filler class. To be more precise: the instances the role filler class, instances of the role class are connected to, are a subset of all instances of the role filler class. In this case, association specialization would imply the constraint that only those instances of a role class may be connected to instances of the associated class that are connected to instances of role filler class that participates in the generalized association.

RC24 It should be possible to express a specialization of an association as an association between the same classes that are part of the generalized association.

Rationale: There is need for this specific kind of dependency. Providing a corresponding concept with the language contributes to productivity and integrity.
Priority: medium

RC25 If the specialization of associations demands for subclasses of classes involved in the generalized association, the use of delegation should be supported. That means that the following constraint is enforced: only those instances of a role class may be linked to

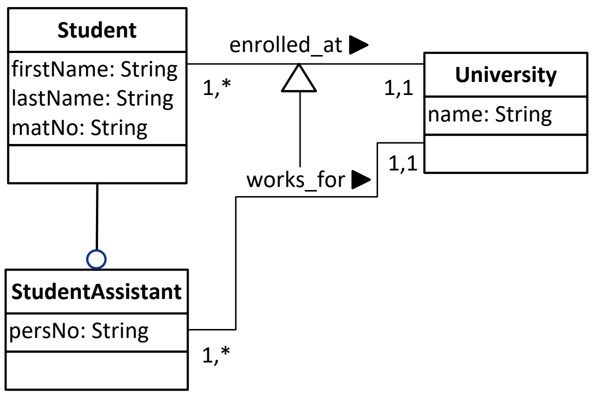

Figure 2.15: *Association Specialization and Delegation*

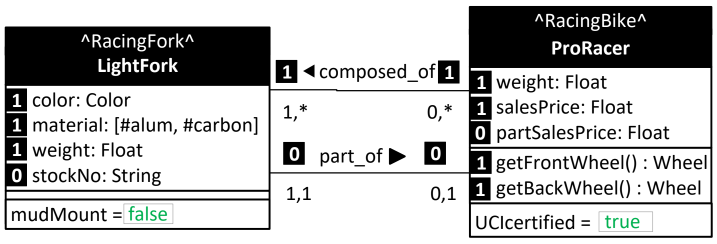

Figure 2.16: *Illustration of Dependency from an Association on a Higher Level*

instances of the associated class the corresponding instance of the role filler class are linked to.

Rationale: Supporting this specific kind of association specialization should support modelling convenience and model integrity.
Priority: medium

In multi-level systems, dependencies may exist between associations on different levels. The example in Fig. 2.16 shows two intrinsic associations between a class representing types of professional racing bicycles and a class that represents types of light weight racing forks. On M_1, that is on the level of particular types, a fork type may be associated with multiple bicycle types, et vice versa. On M_0, each object representing a particular racing bike linked to exactly one object representing a particular fork. Apparently, the existence of these two associations recommends the interpretation that only such a fork can be mounted to a bike the type of which is in the set of fork types associated with the type of the bike. This is indeed the intended semantics of the shown intrinsic associations. However, it is not sufficiently defined by two associations with one association on one level higher than the other one. It is not unambiguously defined that the association **`part_of`** corresponds to the higher level association **`composed_of`**.

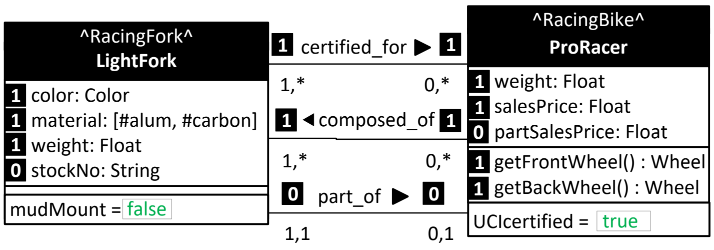

Figure 2.17: *Illustration of Ambiguity Related to Associations in Multi-Level Models*

The slightly modified example in Fig. 2.17 makes this ambiguity more obvious. Without further constraints, the formal semantics do not define clearly whether **`part_of`** corresponds to **`composed_of`** or to **`certified_for`**. At the same time it also illustrates the need for association specialization on higher levels of classification. Apparently, it would make sense to specify **`composed_of`** as a specialization of **`certified_for`**.

RC26 The new version of the FMML^x should allow for defining dependencies between associations on different classification levels. In particular, it should be possible to define an association on level m as a refinement of an association on level m+1.

Rationale: This concept allows preventing ambiguities arising from intrinsic associations (see example in Fig. 2.17).
Priority: high

2.2 Analysis and Management of Models

Management and analysis of a model depend chiefly on the information represented in a model. In the current version, model management was not accounted for because it is not part of the core language specification. However, with respect to the design of modelling tools, management and analysis of models are important. To support the analysis and management of models, concepts are required that, among others, enable sophisticated (statistical) analysis on models, and support advanced model navigation as well as queries on models. Furthermore, there should be concepts that allow for user management, backtracking of a model's history, and the management of model versions.

2.2.1 Model Analysis

The Xmodeler does not only feature a complete programming language (XOCL) but also a comprehensive extension of classes with numerous methods that facilitate elaborate model analysis. The following examples show how to create reports on a model **`Products`**:

Object	*Operation*	*Returns*
Model	averageNumOfAttributes()	average number of attributes per class without inherited attributes
Model	averageNumOfAllAttributes()	average number of attributes per class including inherited attributes
Model	averageNumOfIntrinsicAttributes()	average number of attributes per class without inherited attributes
Model	highestClassificationLevel()	highest classification level of all classes
Model	averageClassificationLevel()	average classification level of all classes
Model	allDependentModels()	all models that depend on the model
Class	allOf()	all classes and meta-classes of a class up to the root class
Class	allTransInstances()	all instances of a class and all instances of instances down to M_0
Class	intrinsicAttributes()	intrinsic attributes of a class without inherited attributes
Class	allIntrinsicAttributes()	intrinsic attributes of a class with inherited attributes

Table 2.4: *Examples of Operations to Support Model Analysis*

```
Products.classes.size()    returns the number of classes
d:=0; Products.classes->collect( c | d:= d + c.attributes.size());
      returns number of all attributes
```

While XOCL is powerful, more complex analysis scenarios require a considerable amount of coding. Providing operations that address relevant analysis scenarios with the language would contribute not only to making model analysis more convenient, but also more reliable. The current version of XCore and the FMML^x offer already a few operations of this kind, e.g., **`allInstances`** that delivers the set of all instances of a class. Multi-level models are more complex than one-level models, which demands for additional types of analysis. The operations in Table 2.4 illustrate how model analysis could be supported with dedicated operations. Note that the list is not comprehensive.

In addition to operations that address particular analysis needs, it would be helpful to simplify the use of XOCL for retrieval purposes through the specification of a more focused retrieval language that accounts for peculiarities of multi-level models. The following queries serve to illustrate the idea through a few examples. That does not mean that the required query language should feature a corresponding concrete syntax.

```
all objects of <class name> where <attribute name>.value > value
all classes where <attribute name> = String
aggregate all <attribute name>.value of <class name>
average of all <attribute name>.value of <class name>
standard_devitation of all <attribute name>.value of <class name>
```

```
all classes that implement <operation name>
all operations that send message <operation name>
```

RT1 The language should include operations for advanced model analysis. That comprises the creation of model statistics including those relevant for multi-level models.

Rationale: Writing code to analyse models will often be regarded as too time-consuming. Adding respective operations as a default interface to models and to all classes makes model analysis more convenient and more reliable.
Priority: high

RT2 The language should be supplemented by a specific query language that allows querying all properties of a multi-level model and that is clearly easier to use than the XOCL.

Rationale: Queries on multi-level models are of crucial importance not only with regard to traditional model analysis, but also for retrieving objects and object values on all classification levels covered by a multi-level model.
The upcoming, multi-level version of the XOCL will allow querying all aspects of a multi-level model. However, it may be useful to define a more specific language, as well as additional tools that enable more convenient queries.
Priority: medium

2.2.2 Support for Changes

Conceptual models evolve over time. Corresponding changes are not a problem as long as they are monotonic and do not produce any side-effects, such as adding an attribute to a class in a traditional object-oriented model. All other cases require specific support by a modelling tool to prevent inconsistencies. Since multi-level modelling introduces additional dependencies, changes are clearly more challenging than in the traditional paradigm. Even adding a property may create a problem. If, for example, an intrinsic attribute is added to a class, that may have an effect not only to its direct instances, but also to instances of its instances. But even if a regular property is added, this may require major changes (see example in Fig. 3.2). A complete analysis of possible changes and approaches to handle them goes beyond the scope of this report. Therefore, only a few selected requirements will be proposed here.

RT3 A modelling tool should support adding properties, both regular and intrinsic. This includes accounting for possible side-effects.

Rationale: Adding properties is a regular operation during the development of a model.
Priority: high

RT4 A modelling tool should support deleting properties, both regular and intrinsic. This includes accounting for possible side-effects.

Rationale: Deleting properties is a regular operation during the development of a model.
Priority: high

RT5 A modelling tool should support the modification of properties, both regular and intrinsic. Modifications relate to the name of a property as well as its specification. Again, possible side-effects need to be accounted for, too.

Rationale: The modification of properties is a regular operation during the development of a model. While it could be realized through deleting a property in the original class and adding it in a new class, it is clearly more convenient to provide moving as a single operation.
Priority: high

RT6 A modelling tool should support moving properties, both regular and intrinsic, from one class within a hierarchy to another one.

Rationale: Moving properties from one class within a hierarchy to another one is an operation that is required occasionally.
Priority: high

Note that new requirements such as contingent instantiation levels of properties (RC2, RC3), multiplicities of attributes (RC18), or deferred instantiation of attribute classes (RC19) substantially increase the complexity of changing properties.

RT7 A modelling tool should allow for changing an object's classification level without compromising the integrity of a model.

Rationale: During the development of a model the classification hierarchy may have to be modified. That includes deleting levels or inserting classes between existing levels.
Priority: high

Again, new requirements such as the demand for contingent level classes (RC1) add complexity to this kind of change operations.

Challenge: Changes applied to multi-level models may have multiple effects that may not be directly obvious. Some of these effects can be handled automatically by a tool, others require user input. In any case, a modelling tool needs to cover all possible changes appropriately. That requires a more specific requirements analysis and the thorough design and implementation of operations that handle changes.

2.2.3 Model History

Sometimes it is important to know previous states or versions of a model or of model elements respectively. At best, it would be possible to roll back to a previous state of a model. There are different options to support a rollback to a previous version. All states of every model element are stored together with a time stamp that defines the time of their creation. If, for example, an attribute of a class is changed, a new version of the class would be created. Instead of storing the entire model element, only those parts could be stored that were actually changed. In that case, a new version of the attribute would be created, and the old one would be stored. In addition, it is possible that any state change is recorded, or that only those states/versions are stored that were explicitly marked as such.

RT8 The language should allow for adding meta data that is relevant for storing a model's history.

Rationale: This is the prerequisite for keeping track of previous model states.
Priority: high

RT9 A modelling tool should allow for explicitly defining model versions and support the rollback to previous versions.

Rationale: The explicit definition of versions enables the modeller to define those states that represent a relevant step in the evolution of a model. As a consequence, browsing through the history of previous model states should be more clearly and more convenient.
Priority: high

Alternatively, it is conceivable that the state of a model is stored at a certain frequency, e.g., every 30 seconds. However, in any case, modelers should be provided with the option to define a particular state as a version.

RT10 A modelling tool should support the rollback to previous versions.

Rationale: This is an important feature in case one wants to reconstruct the evolution of a model or to continue with a previous version after it turned out that it is more appropriate than the current version.
Priority: medium

RT11 Modelling tools should provide support for detecting the delta of two model versions.

Rationale: This feature helps to analyse the evolution of a model and to compare two versions in particular.
Priority: medium

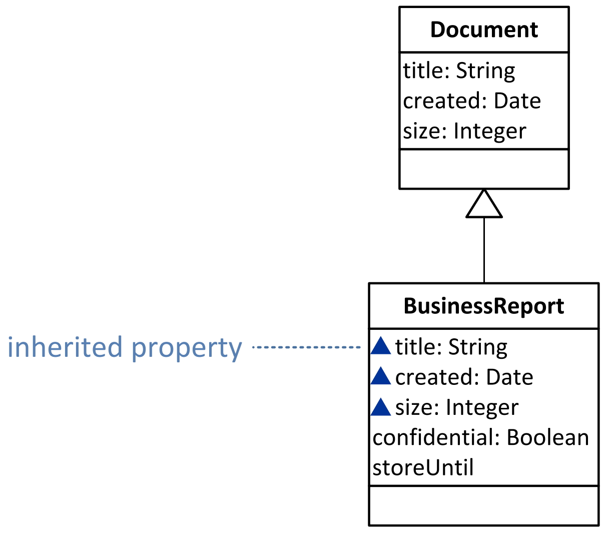

Figure 2.18: *Illustration of How to Represent Inherited Properties*

2.2.4 Information Filtering

Models can become complex. Therefore, a tool should allow for reducing the complexity of their representation. A common approach to accomplish this is providing filters, which allow to fade out and fade in information upon request. The current version of the Xmodeler provides information filtering already, e.g., to hide or show class properties. The specific features of the FMML^x demand for additional filters, which are addressed by the following requirements. The rationale for all these requirements is basically the same, namely to allow for adapting the complexity of a representation to users' needs.

RT12 It should be possible to fade in/out intrinsic properties such as intrinsic attributes, operations and associations.

Rationale:
Priority: medium

RT13 It should be possible to fade in/out inherited properties such as intrinsic attributes, operations and associations. As a default, inherited properties are not shown. However, it can be useful to present them. In that case, inherited properties should be marked as such. See illustration of possible realization in Fig. 2.18.

Rationale:
Priority: medium

RT14 It should be possible to fade in/out operations that are available through delegation. In case these operations are shown, they should be marked as such. See illustration of possible realization in Fig. 2.19.

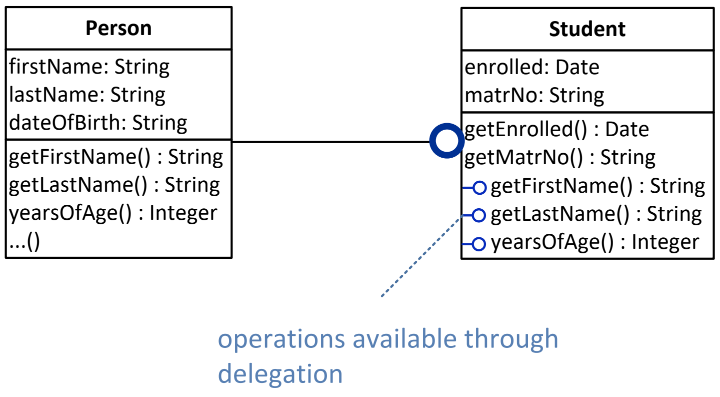

Figure 2.19: *Illustration of How to Represent Operations Available Through Delegation*

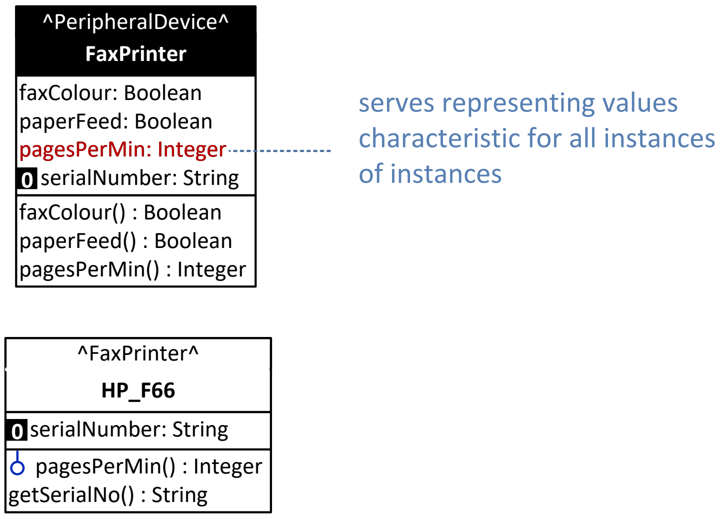

Figure 2.20: *Illustration of How to Represent Operations Available Through Delegation to Class*

Rationale:
Priority: medium

RT15 It should be possible to fade in/out operations that are available through delegation to class. If these operations are shown, they should be marked as such. See illustration of possible realization in Fig. 2.20.

Rationale:
Priority: medium

In general, a tool should allow for the definition of representation preferences in a user profile. However, it should be possible to modify visibility when working with a model.

2.2.5 Compensation for Lack of Static Typing

The FMML^x is an executable language. This is because it is an instance of Xcore and embedded in the Xmodeler. The flexibility of this integrated meta-modelling and meta-programming environment results in part from the choice of dynamic typing. It does not only facilitate a powerful implementation of delegation, but supports versatile introspection, too. While the use of the FMML^x benefits from dynamic typing, the lack of static typing has its downside, too, because it reduces the information content of code. The following example queries illustrate this shortcoming.

```
Search for all classes, the instances of which send the message m1
    to objects of the class c.
```

The analysis of code allows identifying all operations and, hence, all classes, that implement sending a message by the signature of m1. However, the code does not allow to unambiguously determine whether the message is sent to an instance of class c.

```
Search for all classes, the objects of which receive messages from
    objects of the class c.
```

This is an extended version of the previous query, since it requires collecting all messages sent by objects of the class c.

RT16 A modelling tool that features dynamic typing should compensate for the lack of static typing. This could be achieved by adding information to code.

Rationale: If the representation of code is ambiguous, the analysis of large models can be painful and produce serious threats to system integrity.
Priority: medium

2.2.6 Model Completion

Since the FMML^x is executable, the specification and implementation of operations is of significant relevance. The creation of a model includes various routine tasks. This is the case, too, for mapping a model to code. A typical example of routine tasks is the creation of access operations.

That includes the definition of operation interfaces according to conventions (see examples in Figures 2.21, 2.22, 2.23, and 2.24), the creation of code and the realization of links to the elements that are subject of an access operation (see Fig. 2.23)

<getter> <attribute name> ‘():’ <attribute class/type name> ‘()’

<getter> := ‘get’

Example: *getFirstName(): String*

<setter> <attribute name> ‘(‘ <paramName> ‘:’ <name of attribute class> ‘)’

<setter> := ‘set’

Example: *setFirstName (n: String)*

Figure 2.21: *Construction of Access Operations for Attributes with Multiplicity* `1..1`

<remover> <attribute name> ‘():’ < name of attribute class> ‘()’

<remover> := ‘removeFrom’

Example*: removeFromColors(): Color*

<adder> <attribute name> ‘(‘ <paramName> ‘:’ < name of attribute class> ‘)’

<adder> := ‘addTo’

Example*: addToColors (c: Color)*

Figure 2.22: *Construction of Access Operations for Attributes with Max. Cardinality* > 1

<getter> <name of pointer variable> ‘():’ < name of associated class> ‘()’

<getter> := ‘get’

Example*: getDepartment(): Department*

<setter> <name of pointer variable> ‘(‘ <paramName> ‘:’ < name of associated class> ‘)’

<setter> := ‘set’

Example*: setDepartment (d: Department)*

Figure 2.23: *Construction of Access Operations for linked Objects with Multiplicity 1..1*

```
<remover> <name of pointer variable> ‘():’ < name of associated class> ‘()’

<remover> := ‘removeFrom’

    Example: removeFromDepartments(): Department

<adder> <name of pointer variable>  ‘(‘ <paramName> ‘:’ < name of associated class> ‘)’

<adder> := ‘addTo’

    Example: addToDepartments (d: Department)
```

Figure 2.24: *Construction of Access Operations for linked Objects with Max. Cardinality* > 1

RT17 A modelling tool should support the generation of access methods according to adaptable policies.

Rationale: Generating access operations decreases development costs and contributes to comprehensibility and integrity of models and software.
Priority: high

2.2.7 User Management

Often, models will be developed an used by more than one person. Therefore, a modelling tool should support some kind of user management. That includes the creation and management of profiles and assigning modifications of models to users.

RT18 A modelling tool should support the creation of user profiles. A user profile stores user's visualization preferences, which include font sizes, colour, the layout of window frames, etc.

Rationale: Adapting the representation of a model to individual preferences allows creating a more convenient and possibly more efficient environment. It is an unnecessary burden for users, if they have to specify their preferences each time they use a model (or an instance of the tool) that is used by others, too.
Priority: medium

RT19 With respect to keeping track of the evolution of a model, a tool should also store the users that were responsible for a certain modification.

Rationale: Sometimes, it is useful to find out who made certain modelling decisions, especially, if they seem unclear or inappropriate.
Priority: medium

Further aspects of user management could address the definition of access rights. Access rights could be defined for an entire model, for certain aspects of a model, or for certain operations on models. Currently, we do not intend to support multi-user editing of models. Therefore, these aspects of user management are of low priority only.

2.3 Concrete Syntax

The traditional distinction between general-purpose modelling language (GPML) and domain-specific modelling language (DSML) cannot be directly applied to the FMML^x . On the one hand, the FMML^x is a meta modelling language, since is supposed to be used for the specification of any kind of language. That would qualify it as a GPML or, more precisely, as a general purpose meta-modelling language. Only the languages that are specified with the FMML^x would then be domain-specific. On the other hand, however, the FMML^x does not have to be restricted to a meta-modelling language. Instead, it can be used to create multi-level models that also include domain-specific concepts – usually on lower levels of classification. As a consequence, it is conceivable to use different notations with the FMML^x . On more generic, that is, higher levels of classification, one could use a notation that lacks any reference to specific domain concepts. On lower levels, notations could be used that refer to common visualizations of domain-specific concepts. However, even though the lower levels could be modelled with the FMML^x , it is more appropriate to leave them to domain-specific languages, which in turn would be specified with the FMML^x . Therefore, the notation of the FMML^x is generic, that is, it does not account for a specific visualization of domain-specific concepts.

It is challenging to develop requirements for the (generic) concrete syntax of a multi-level modelling language. This is for two main reasons. First, by their very nature, the concepts of such a language are not only rather abstract, but they also go beyond concepts of traditional object-oriented modelling. Second, the preferences for certain visualizations are not independent of previous experiences. The group of researchers that is familiar with the previous version of the FMML^x , and that has been involved in its development, is probably biased, both with respect to the concepts it offers and its notation. Researchers that were involved in the development of other languages for multi-level modelling are likely to be biased, too. Therefore, the following requirements will reflect subjective preferences. The only way to relax the effect of subjective preferences is to aim at comprehensible justifications of requirements, e.g., by refering to more general principles.

RN1 The notation should be in line with widespread notations of conceptual modelling languages. That recommends to use shapes for the representation of classes similar to those used with the UML. This is especially the case for the representation of classes on M_1, because they are typically subject of traditional object models.

Rationale: The use of familiar representations is likely to promote the understandability of multi-level models.
Priority: high

RN2 The notation should build on that of the previous version of the FMML^x and account for critical feedback on the previous notation at the same time.

Rationale: The previous notation resulted from a long process of developing, using, and improving multi-level modelling languages. Therefore, it seems reasonable to assume that it is regarded as widely appropriate by those who used it. At the same time, certain aspects of the notation were subject of critical comments that should be accounted for.
Priority: medium

RN3 The additional concepts defined with the new version of the FMML^x , e.g., contingent-level classes, should be represented clearly.

Rationale: This requirement is a direct consequence of the general demand for understandability of models.
Priority: high

RN4 The representation of classification levels should follow certain design guidelines.

Rationale: In the previous version, the colours used for distinguishing classification levels were chosen in a widely arbitrary manner, which made it difficult to memorize what level a specific colour represented.
Priority: medium

A possible approach to satisfy this requirement could be to use colours according to their position in the colour spectrum. In addition, it would be helpful, if the corresponding grey-scale representations were characterized by a step-wise increasing lightness. With respect to requirement RN4 and the long-standing use of black as background colour of classes on M_1, the application of the colour spectrum would start on M_2 only. Note that this requirement would, in part, be in conflict with requirement RN4.

3 Terminology

It is the purpose of a technical terminology to promote effective and efficient communication. Therefore, the concepts it provides should adequately narrow the range of possible interpretations and allow for expressing relevant differences. Multi-level modelling can be seen as new paradigm, both for conceptual modelling and for software development in general. It changes the traditional approach to object-oriented modelling in a way that requires modellers to rethink the way how they perceive and conceptualize a domain, and what they learned about modelling languages. The most important sign of a new paradigm is the fact that the terminology of a previous paradigm is not sufficient and/or misleading. Based on an analysis of these terminological challenges, I will propose a new terminology that is suited to cope with the peculiarities of multi-level modelling.

3.1 Misleading Application of Current Terminology

The concepts class and object are at the core of the object-oriented paradigm. It makes sense to use them for multi-level modelling, too, because they serve a similar purpose. However, they have to be extended. First, in multi-level models, every class is an object at the same time. Second, the classification level of classes is not limited. The conceptualization of classes in multi-level systems is, however, different, because they allow for properties that are not defined for direct instances only. This feature has a substantial impact on two terms that are also essential for the traditional paradigm, classification and instantiation. In the traditional paradigm, a class can be conceptually regarded as an abstraction that specifies the extension of all objects of the same kind. The types of its properties, that is, of its attributes and the classes it is associated to, define this extension. Hence, instantiation means to select an element out of this set. As a consequence, the instances of a class cannot be further instantiated. This is different for classes in multi-level systems. Since a class may have intrinsic properties, the extension of its direct instances is not defined by the carthesian product of the sets specified with the types of its attributes or associated classes. Instead, intrinsic properties are inherited to its instances, apart from those that are to be instantiated on the level below. Not only that instantiation in multi-level systems can be different from instantiation in traditional object-oriented systems, it also seems to violate an iron law of the traditional paradigm, that is, the strict dichotomy of instantiation and specialization. On closer inspection, multi-level modelling does not allow for

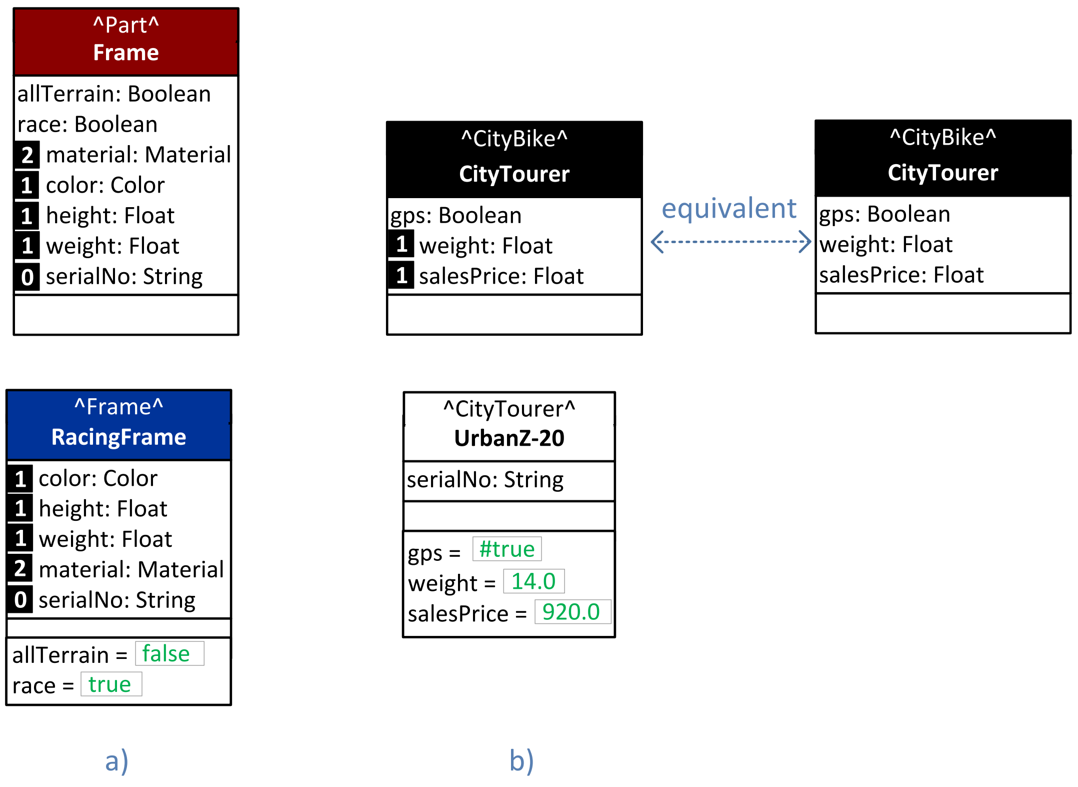

Figure 3.1: *Examples of Instantiation with Intrinsic Attributes*

applying instantiation and specialization simultaneously, at least as long as specialization is regarded to satisfy the substituability constraint (Liskov and Wing 1994). Instead, it allows for combining instantiation and inheritance. Inheriting properties is a prerequisite of deep instantiation.

Since classification and instantiation in multi-level modelling are not equivalent to the corresponding terms in the traditional paradigm, it is misleading to apply these terms to multi-level modelling without differentiation. Example a) in Fig. 3.1 shows the instantiation of a class in a multi-level system. The class **`Frame`** is located on M_3. Its regular attributes are instantiated and initialized within the instance **`RacingFrame`** on M_2, while the intrinsic attributes are inherited to **`RacingFrame`**. Example b) shows a similar constellation. However, all intrinsic attributes are to be instantiated with the direct instances of the class **`CityTourer`**. Therefore, the class could equivalently be specified without intrinsic attributes. Nevertheless, it is different from an instantiation in the traditional paradigm, because an attribute is added to the instance.

The example in Fig. 3.2 illustrates a specific challenge related to classification and instantiation in multi-level systems. The instantiation of the class **`CityBike`** on M_3 should not qualify as an instantiation in a multi-level system, because not a single attribute is instantiated with would-be instances such as **`ElectricCityBike`**. Therefore, **`ElectricCityBike`** should rather be specialized from **`CityBike`**, which recommends assigning them to the same classification

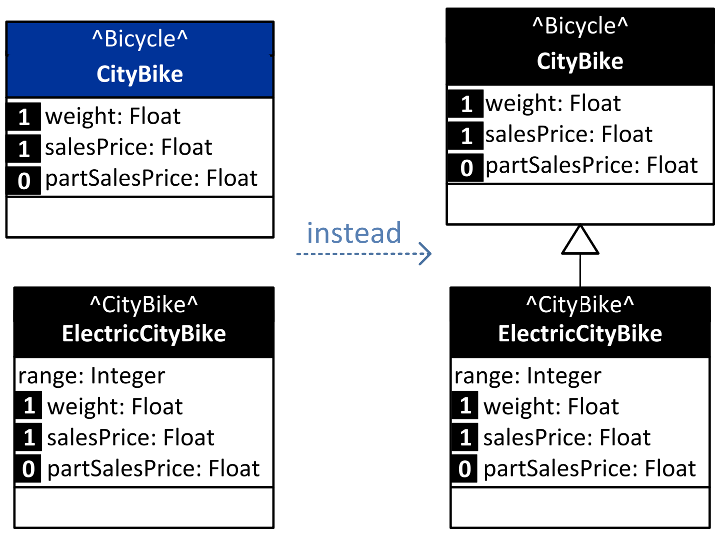

Figure 3.2: *Instantiation Replaced by Inheritance*

level, which is M_2 in the example. However, the specification of a class may change over time. If **`CityBike`** is extended later by a regular attribute such as **`integratedLock`**, which should be instantiated on M_2, specialization would not be adequate any longer. Therefore, this example underlines the need for a tool to support extensive, and challenging, change operations.

To avoid confusion, a terminology for multi-level modelling requires specific concepts of class, object as well as of instantiation and classification. As a consequence, the notion of classification level needs to be adapted to the peculiarities of multi-level modelling, too. Furthermore, the semantics of inheritance needs to be adapted, since inheritance relationships can be defined between classes on different levels of classification.

3.2 Limitations of Current Terminology

The concepts used in the traditional paradigm are not sufficient to directly represent ideas of multi-level models. A major reason for this restriction is the fact that instantiation and classification may evolve over multiple levels without being transitive. This is different from generalization/specialization hierarchies. If we, for example, want to denominate the set of classes that are part of the entire specialization hierarchy below the class c, we could use an expression like “all subclasses of c”, since every class in the hierarchy, no matter on what level, would qualify as a subclass of c.

The existing terminology would not allow to directly express sets of objects that were instantiated across multiple levels. It is even extremely cumbersome to describe such as set

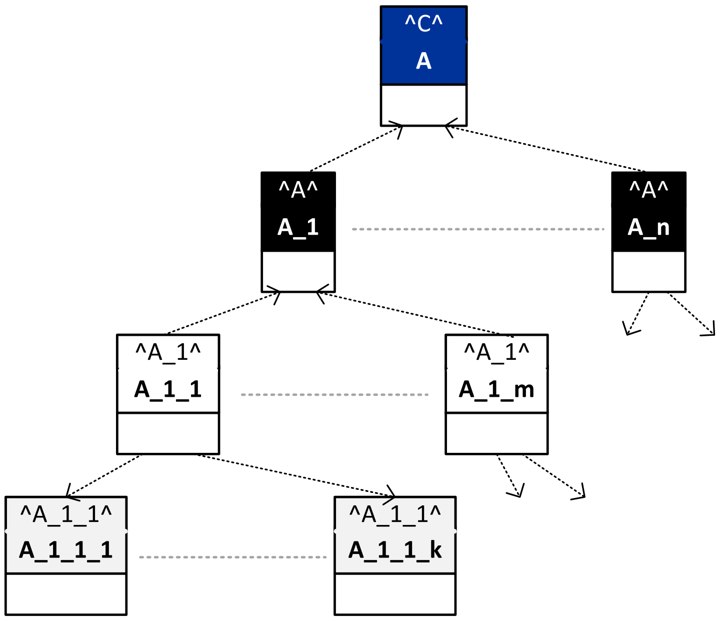

Figure 3.3: *Illustration of Relaxed and Restricted Concept of Instantiation*

appropriately. Take, for example, the hierarchy in Fig. 3.3. The hierarchy classes below the class **A** are all instances of **A** plus all instances of all instances of **A** plus all instances of all instances of all instances of **A**. Obviously, the complexity of expressions like these is a serious obstacle to communication.

A further, less demanding problem related to the terms class and object. Every class is an object, but not every object qualifies as class. In the traditional paradigm, objects that are not classes at the same time, can usually be referred to as objects, because classes are not regarded as objects, or as instances. Both options do not work for multi-level systems. However, it is cumbersome to use an expression like "objects that cannot be instantiated". At the same time, it would be ambiguous, to, because it holds for abstract classes, too.

Other terminological issues relate to the ambiguity of "meta". In the old paradigm it usually serves to name classes on M_2. In multi-level modelling, every class above M_1 may qualify as a meta-class. Therefore, the term is unambiguous only in relation to a particular level.

3.3 Proposal for Specialized Terminology

The description of the proposed terminology is not intended to be comprehensive, because concepts that are known from the traditional paradigm are not defined anymore.

Class: A class is an object. It may have superclasses. If a language allows for multiple inheritance, it may have many superclasses. Otherwise, it must not have more than one superclass.

Note that the restriction to one superclass may be relaxed, if the language architecture makes use of a core (meta-) class that serves as a transparent superclass for all classes in a model. A class is defined through properties. A property is either an attribute, an association, or an operation.

Intrinsic Property: An intrinsic property is specified by an instantiation level that defines the level where it is to be instantiated. The intended instantiation level has to be lower than $l - 1$, where l is the level of the class where the intrinsic property is first defined.

Inherited from Root Class: This term refers to RC10. Such a property is implicitly inherited from a root or core class that serves as a superclass of all classes in a multi-level system. For example, an attribute **`name`** is usually needed in every class. A further example would be an operation like **`allInstances()`**.

Object: Every object is of exactly one class. It has a state. The state of an object is defined by its properties (in case, the object is a class), the objects or values instantiated from the classes or types of its attributes, and references to external objects, which are instances of classes, the object's class may be associated with. Usually, but not necessarily, objects enforce encapsulation. From a conceptual perspective, that means that objects and values that where instantiated from attribute classes or types, do not have an identity independent of the object.

Intrinsic Instantiation: To avoid misinterpretation, the term "intrinsic instantiation" can be used instead of "instantiation". It represents the creation of an instance from a class, which, in addition to the instantiation of properties, may also include inherited intrinsic properties. A more precise definition of intrinsic instantiation is subject of the language specification. In particular, it needs to be decided whether it should be possible to add properties to an instance (if the instance is a class). The examples in Fig. 3.4 illustrate these options. The intrinsic instantiation a) is restricted to properties specified in the (meta) class. In the instance, the property types are instantiated or, in case of intrinsic properties, inherited from the (meta) class. With example b), the instantiated class adds a further property. While that clearly adds flexibility, it is doubtful whether it should be allowed for. It is an essential aspect of classification that a class defines the extension of its instances. That would be still the case with option a). In case of option b), however, an indefinite number of properties could be added. Hence, the class would no longer specify the extension of its instances. Example c) shows how the intended class in b) could be specified by specializing the instance in a).

Intrinsic Classification: The term "intrinsic classification" can be used instead of "classification". It represents the creation of a class that represents a set of objects that are of the same kind. In addition to abstracting possible object states to class properties, intrinsic classification may also comprise the generalisation of object properties (in case the object is a class).

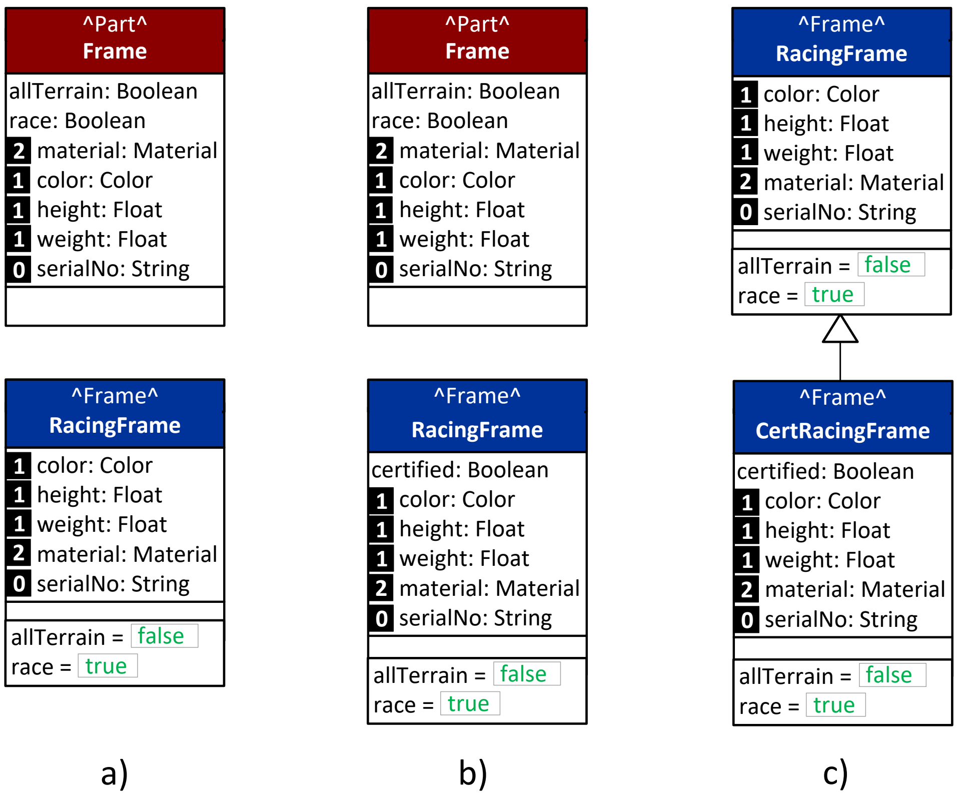

Figure 3.4: *Illustration of Flavours of Intrinsic Instantiation*

Level: The level of an object serves to define how many times it can be (intrinsicly) instantiated. To avoid misinterpretation, the levels should not be indicated with an "M" as in the traditional paradigm, but with "ML" instead. Furthermore, the integer that indicates the level should be print in subscript to improve readability.

Specialization: A class can be specialized into more specific classes, which means that it inherits all its properties to its subclasses. Its state is not inherited. Specialization is restricted to classes on the same level. Note that this restriction does not apply to inheriting properties from a root class.

Generalization: Classes can be generalized to superclasses. In case of single generalization a class must not have more than one superclass. If a language (such as the FMML^x) allows for multiple generalization, it may have multiple superclasses. Generalization is the inverse operation to specialization.

Basic Object: An object that is located on level ML_0 can be referred to as "basic object". It must not have properties and cannot be instantiated.

Abstract Class: An abstract class can exist on every level above ML_0. It must not be instantiated. Its sole purpose is to allow for a generalization over similar classes.

Contingent Level Class: A contingent level class does not have a definite level. Its class must be contingent, too. It can be intrinsicly instantiated to contingent classes or to regular objects on every level. It depends on the chosen notion of intrinsic instantiation whether the level of objects a contingent class can be instantiated to depends on its properties. If intrinsic instantiation of a class is restricted to inheritance of intrinsic properties and the instantiation of objects or values from properties of that class, the highest level of an object an contingent level class can be instantiated to, would be determined by the highest instantiation level of its intrinsic properties. If that option is chosen, contingent level classes must include contingent level properties. Otherwise, their level would not be contingent, but determined by their properties.

Contingent Level Property: A contingent level property does not have a unique instantiation level. Its instantiation level can be any level p with $p < l - 1$, where l is the level of the class that defines the property.

Meta Class on Level: The term "meta class of c on level n" is used to refer to the class on $\mathrm{ML}_{\mathbf{n}}$ in the hierarchy of meta classes of a class c. The direct class of c would be referred to as "class of c".

Hierarchy of Classes: The term "hierarchy of classes of c" is used to refer to the list of (meta-) classes of c.

Instance Tree: All instances and indirect instances of a class c are referred to as "instance tree of c". All direct instances are called "all instances" or "all intrinsic instances".

4 Conclusions

The design of any complex artefact demands for a thorough analysis of requirements. In case of a modelling language requirements are harder to identify than for application systems. This is even more so for a multi-level modelling language, because it is part of a new paradigm. Not only that few users are aware of this new paradigm. In addition, the ideas of how to evolve it are still not consolidated. Against this background, the analysis of requirements for multi-level modelling languages is a remarkable challenge. Clearly more than with any other requirements analysis, it cannot be expected to be complete. Furthermore, it may turn out that some of the presuppositions that motivated requirements need to be revised. However, at the same time, this situation creates also an opportunity. To further promote the research field of multi-level modelling, it is important to consolidate and eventually unify foundational concepts. The requirements presented in this report are also an attempt to contribute to a discourse on the future of multi-level modelling. It seems more promising to start the competition not only with the presentation of language specifications, but during requirements analysis already. First, the more researchers are involved in the analysis of requirements, the more likely it is to identify relevant issues. Second, from a psychological perspective, it seems to be easier to develop a consensus on requirements than on the comparative evaluation of existing artefacts, at least as long as the creators of these artefacts are involved.

Bibliography

Atkinson, Colin and Ralph Gerbig (2016). "Flexible Deep Modeling with Melanee". In: *Modellierung 2016, 2.-4. März 2016, Karlsruhe - Workshopband*. Ed. by Stefanie Betz Ulrich Reimer. Vol. 255. Modellierung 2016. Bonn: Gesellschaft für Informatik, pp. 117–122. ISBN: 978-3-88579-649-7.

Atkinson, Colin, Bastian Kennel, and Bjoern Goss (2011). "The Level-Agnostic Modeling Language". In: *Software Language Engineering*. Ed. by Brian Malloy, Steffen Staab, and Mark van den Brand. Vol. 6563. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 266–275. ISBN: 978-3-642-19439-9.

Atkinson, Colin and Thomas Kühne (2001). "The Essense of Multilevel Metamodeling". In: *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Ed. by Martin Gorgolla and Cris Kobryn. Vol. 2185. Lecture Notes in Computer Science. Berlin and London, New York: Springer, pp. 19–33. ISBN: 978-3-540-42667-7.

Atkinson, Colin and Thomas Kühne (2008). "Reducing accidental complexity in domain models". In: *Software & Systems Modeling* 7.3, pp. 345–359. ISSN: 1619-1366. DOI: `10.1007/s10270-007-0061-0`. URL: `http://homepages.ecs.vuw.ac.nz/~tk/publications/papers/accidental-springer-online.pdf`.

Brooks, Frederick P. (1995). *The Mythical Man Month: Essays on Software Engineering*. 2nd ed. Reading, MA: Addison-Wesley. ISBN: 978-0-201-83595-3.

Clark, Tony, Paul Sammut, and James Willans (2008). *Applied metamodelling: a foundation for language driven development*. URL: `https://eprints.mdx.ac.uk/6060/1/Clark-Applied_Metamodelling_%28Second_Edition%29%5B1%5D.pdf`.

Clark, Tony and James Willans (2012). "Software Language Engineering with XMF and XModeler". In: *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. Ed. by Marjan Mernik. IGI Global, pp. 311–340.

Frank, Ulrich (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, Ulrich (2016). “Designing Models and Systems to Support IT Management: A Case for Multilevel Modeling”. In: *MULTI 2016 – Multi-Level Modelling. Proceedings of the Workshop in Saint-Malo*, pp. 3–24. URL: http://www.wi-inf.uni-due.de/FGFrank/documents/Konferenzbeitraege/ML-ITML-Multi2016.pdf.

Jeusfeld, Manfred A. (2009). “Metamodeling and method engineering with ConceptBase”. In: *Metamodeling for Method Engineering*. Ed. by Manfred A. Jeusfeld, Matthias Jarke, and John Mylopoulos. Cambridge: MIT Press, pp. 89–168. ISBN: 978-0262101080.

Kaczmarek-Heß, Monika (2017). “Multilevel Model of Events in Support of Enterprise Agility in the Realm of Enterprise Modeling”. In: *Proceedings of the CBI 2017*, pp. 267–276.

Kaczmarek-Heß, Monika and Michael Heß (2018). “A Multilevel Model of Pharmaceuticals”. In: *Multikonferenz Wirtschaftsinformatik 2018 (MKWI 2018)*, pp. 1516–1527.

Kaczmarek-Heß, Monika and Sybren De Kinderen (2017). “A Multilevel Model of IT Platforms for the Needs of Enterprise IT Landscape Analyses”. In: *Business & Information Systems Engineering* 59.9, pp. 315–329. URL: https://link.springer.com/article/10.1007/s12599-017-0482-4.

Kühne, Thomas and Daniel Schreiber (2007). “Can programming be liberated from the two-level style: multi-level programming with deepjava”. In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA '07)*. Ed. by Richard P. Gabriel et al. Vol. 42,10. ACM SIGPLAN notices. New York: ACM Press, pp. 229–244. ISBN: 978-1-59593-786-5. URL: http://atlas.tk.informatik.tu-darmstadt.de/Publications/2007/p229-kuehne.pdf.

Lara, Juan de and Esther Guerra (2010). “Deep Meta-modelling with MetaDepth”. In: *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*. Ed. by Jan Vitek. Vol. 6141. Lecture Notes in Computer Science. Springer, pp. 1–20. ISBN: 978-3-642-13952-9. DOI: 10.1007/978-3-642-13953-6_1.

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

Meta Object Facility (MOF) Core Specification: Version 2.0 (2006). URL: http://www.omg.org/spec/MOF/2.0/.

OMG (2010). *OMG Unified Modeling Language (OMG UML), Infrastructure: Version 2.3*. URL: http://www.omg.org/spec/UML/2.3/Infrastructure.

Szyperski, Clemens, Dominik Gruntz, and Stephan Murer (2011). *Component software: Beyond object-oriented programming*. 2nd ed. Addison-Wesley Component software series. London: Addison-Wesley. ISBN: 9780321753021.

Volz, Bernhard Walter (2011). *Werkzeugunterstützung für methodenneutrale Metamodellierung*. Bayreuth.

A Appendix: List of Requirements

RC1

In addition to having classes with a definite level, it should be possible to define classes with a *contingent* level. A contingent level allows for adapting the concrete level of a class to different contexts of use.

Rationale: It happens that two classes on different classification levels could be classified by the same meta-class. However, this can be achieved only, if the meta-class does not have a definite classification level. Instead, its level would have to be contingent with respect to the instantiation context.

Priority: *high*

RC2

It should be possible to define intrinsic attributes with a *contingent* instantiation level.

Rationale: It may not be possible to foresee the concrete instantiation level of an intrinsic attribute, while knowing that it has to be instantiated somewhere down the instantiation line.

Priority: *medium*

RC3

It should be possible to define intrinsic associations and operations with a *contingent* instantiation level.

Rationale: As with intrinsic attributes, it may not be possible to foresee the concrete instantiation level.

Priority: *medium*

RC4

It should be possible to define constraints on the intended instantiation level. These could address a minimum or maximum number of instantiation steps, or a set of alternative instantiation levels, e.g., level 1 or level 2.

Rationale: It is conceivable that at the time of specifying a class on level n, there is not sufficient knowledge available to exactly define an intended instantiation level. But usually at least some restrictions on possible instantiation levels will be known. Hence, the requirements follows from the principle that all knowledge available on a certain level should be specified there.

Priority: *medium*

RC5

It should be possible to define preliminary classes without a meta-class.

Rationale: Sometimes, a bottom-up approach seems to be more appropriate than a top-down approach.

Priority: *high*

RC6

Intrinsic associations should allow the specification of different instantiation levels for both participating classes.

Rationale: It is common in natural language to link two concepts (or object references) on different classification levels in one sentence. Accordingly, it can be a relevant requirement to link two objects on different classification levels in an multi-level model. If the association can be defined on a higher level already, that is, if the association is intrinsic, it follows directly that it must be possible to support different instantiation levels.

Priority: *high*

RC7

The definition of intrinsic associations should imply that corresponding constraints are refined with every instantiation step.

Rationale: By their very nature, intrinsic associations lack certain knowledge about the actual association at the level where they are defined for. At the same time, with every instantiation step, the range of possible options decreases. Hence, refining corresponding constraints dynamically would be an important contribution to system integrity.

Priority: *high*

RC8

The language specification should allow for defining intrinsic associations without specific multiplicities. In other words, it should be possible to defer the specification of multiplicities to a lower level.

Rationale: In cases where the multiplicities of an intrinsic association at the intended association level may vary, the definition of a particular multiplicity would create ambiguity or even a contradiction, hence, a serious threat to a system's integrity.

Priority: *high*

RC9

If absolute minimum or maximum cardinalities are known, they should serve as constraints for the final definition of multiplicities on the intended instantiation level. That means that the actual cardinalities specified on the intended instantiation level have to be within this range. As a consequence, it needs to be possible to mark cardinalities as boundary values.

Rationale: It should be possible to specify all knowledge about multiplicities of associations that is available.

Priority: *high*

RC10

It should be possible to indicate that certain properties of a class are to be inherited to its instances.

Rationale: Inheriting features where it is appropriate enables preventing redundancy.
Note that inheriting properties is different from intrinsic properties. An intrinsic property is not directly instantiated where it is defined, but only at the specified instantiation level. An inherited property can be part of many classes. Therefore, it can be instantiated multiple times. In an ideal case, the implementation of operations can be inherited, too (as it would be the case with the example in Fig. 2.8).

Priority: *high*

RC11

Intrinsic properties must not be explicitly inherited, since they are passed to instances anyway until the intended instantiation level is reached.

Rationale: Explicit inheritance of intrinsic properties would be redundant and would therefore compromise the comprehensibility of a model.

Priority: *high*

RC12

The language should be extended with auxiliary classes that represent specific concepts that are often needed. They include various units of measurement, currencies, multiplicities, and, more specific, economic concepts such as price.

Rationale: The availability of thoroughly defined auxiliary classes contributes to model integrity and modelling productivity.

Priority: *high*

RC13

The definition of specific auxiliary classes should be supported. That includes the definition of classes which represent enumerations or ranges of elements of a certain type/class.

Rationale: This is an incentive for software developers to define specific auxiliary classes and, thus, to improve model and system integrity.

Priority: *high*

RC14

The specification of an association should allow for defining whether the association is bi-directional or uni-directional.

Rationale: Even though this information is required for implementation only, it should be possible to add it to a conceptual model to prepare for a smooth transition to design and implementation. This is even more the case for an executable modelling language like the FMML^x , since there is no clear borderline between models and code anymore.

Priority: *high*

RC15

Delegation should be enabled between classes on any classification level as a specific association between two classes on any level.

Rationale: In object-oriented system, delegation is an important abstraction that allows to compensate for misconceptions of inheritance.
Note that further, more specific, requirements would be needed for a comprehensive specification of delegation. An elaborate analysis of delegation is subject of a further publication that is currently under review. Apart from that, a preliminary implementation of delegation exists already in the current version of the language implementation.

Priority: *high*

RC16

The language should provide generic association types. At best, these types would be instantiated from an association meta-type.

Rationale: Including generic association types promotes modelling productivity and model integrity. Meta-association types provide modellers with more flexibility, since they allow to vary the semantics of generic association types.

Priority: *medium*

RC17

It should be possible to define domain-specific association types. In order to cover the variance of domain-specific association types, it could be helpful, if a language provides meta-association types on a higher level, that would allow the instantiation of meta-association types.

Rationale: The specification of domain-specific association types promotes model integrity.

Priority: *medium*

RC18

It should be possible to specify multiplicities of attributes.

Rationale: Without this feature, it is hardly possible to adequately model cases where attributes are characterized by more than one object of a certain class.
There are not too many cases, where this requirement is relevant.

Priority: *medium*

RC19

It should be possible to specify intrinsic attributes through classes on levels higher than M_1 together with an intended instantiation level of those classes.

Rationale: There are cases where it is not possible to clearly specify the class an intrinsic attribute will have at the level where it is supposed to be instantiated.
Due to our experience with the current version of the FMML^x the lack of deep instantiation of attributes was not a major shortcoming.

Priority: *low*

RC20

The new version should allow for clearly distinguishing between attributes that are regularly instantiated into individual values of instances, that are instantiated into values of instances that actually represent common values of their instances, and those that serve the specification of constraints on attribute values. At the same time, corresponding access operations should be qualified accordingly.

Rationale: These different kind of properties have specific semantics that would be lost, if there was no way to express it somehow, which in turn could compromise the integrity of systems.
This requirement is fairly easy to satisfy and promises clear benefits.

Priority: *high*

RC21

It should be possible to assign categories to operations.

Rationale: Distinguishing categories of operations enables more meaningful model queries and fosters models integrity.

Priority: *high*

RC22

It should be possible to link access operations to the objects they refer to, such as attributes or associations.

Rationale: This additional information would facilitate the updating of operations after relevant aspects of the reference object had been changed.

Priority: *high*

RC23

The new version should support the specification of an association as dependant of another one. That is, the instantiation of the dependant association must not happen without the association it depends on being instantiated, too.

Rationale: Including a corresponding concept in the language does not only free modellers from the specification of a corresponding constraint, it also increases the likelihood that the dependency is explicitly modelled at all.

Priority: *medium*

RC24

It should be possible to express a specialization of an association as an association between the same classes that are part of the generalized association.

Rationale: There is need for this specific kind of dependency. Providing a corresponding concept with the language contributes to productivity and integrity.

Priority: *medium*

RC25

If the specialization of associations demands for subclasses of classes involved in the generalized association, the use of delegation should be supported. That means that the following constraint is enforced: only those instances of a role class may be linked to instances of the associated class the corresponding instance of the role filler class are linked to.

Rationale: Supporting this specific kind of association specialization should support modelling convenience and model integrity.

Priority: *medium*

RC26

The new version of the FMML^x should allow for defining dependencies between associations on different classification levels. In particular, it should be possible to define an association on level m as a refinement of an association on level m+1.

Rationale: This concept allows preventing ambiguities arising from intrinsic associations (see example in Fig. 2.17).

Priority: *high*

RT1

The language should include operations for advanced model analysis. That comprises the creation of model statistics including those relevant for multi-level models.

Rationale: Writing code to analyse models will often be regarded as too time-consuming. Adding respective operations as a default interface to models and to all classes makes model analysis more convenient and more reliable.

Priority: *high*

RT2

The language should be supplemented by a specific query language that allows querying all properties of a multi-level model and that is clearly easier to use than the XOCL.

Rationale: Queries on multi-level models are of crucial importance not only with regard to traditional model analysis, but also for retrieving objects and object values on all classification levels covered by a multi-level model.
The upcoming, multi-level version of the XOCL will allow querying all aspects of a multi-level model. However, it may be useful to define a more specific language, as well as additional tools that enable more convenient queries.

Priority: *medium*

RT3

A modelling tool should support adding properties, both regular and intrinsic. This includes accounting for possible side-effects.

Rationale: Adding properties is a regular operation during the development of a model.

Priority: *high*

RT4

A modelling tool should support deleting properties, both regular and intrinsic. This includes accounting for possible side-effects.

Rationale: Deleting properties is a regular operation during the development of a model.

Priority: *high*

RT5

A modelling tool should support the modification of properties, both regular and intrinsic. Modifications relate to the name of a property as well as its specification. Again, possible side-effects need to be accounted for, too.

Rationale: The modification of properties is a regular operation during the development of a model. While it could be realized through deleting a property in the original class and adding it in a new class, it is clearly more convenient to provide moving as a single operation.

Priority: *high*

RT6

A modelling tool should support moving properties, both regular and intrinsic, from one class within a hierarchy to another one.

Rationale: Moving properties from one class within a hierarchy to another one is an operation that is required occasionally.

Priority: *high*

RT7

A modelling tool should allow for changing an object's classification level without compromising the integrity of a model.

Rationale: During the development of a model the classification hierarchy may have to be modified. That includes deleting levels or inserting classes between existing levels.

Priority: *high*

RT8

The language should allow for adding meta data that is relevant for storing a model's history.

Rationale: This is the prerequisite for keeping track of previous model states.

Priority: *high*

RT9

A modelling tool should allow for explicitly defining model versions and support the rollback to previous versions.

Rationale: The explicit definition of versions enables the modeller to define those states that represent a relevant step in the evolution of a model. As a consequence, browsing through the history of previous model states should be more clearly and more convenient.

Priority: *high*

RT10

A modelling tool should support the rollback to previous versions.

Rationale: This is an important feature in case one wants to reconstruct the evolution of a model or to continue with a previous version after it turned out that it is more appropriate than the current version.

Priority: *medium*

RT11

Modelling tools should provide support for detecting the delta of two model versions.

Rationale: This feature helps to analyse the evolution of a model and to compare two versions in particular.

Priority: *medium*

RT12

It should be possible to fade in/out intrinsic properties such as intrinsic attributes, operations and associations.

Rationale:

Priority: *medium*

RT13

It should be possible to fade in/out inherited properties such as intrinsic attributes, operations and associations. As a default, inherited properties are not shown. However, it can be useful to present them. In that case, inherited properties should be marked as such. See illustration of possible realization in Fig. 2.18.

Rationale:

Priority: *medium*

RT14

It should be possible to fade in/out operations that are available through delegation. In case these operations are shown, they should be marked as such. See illustration of possible realization in Fig. 2.19.

Rationale:

Priority: *medium*

RT15

It should be possible to fade in/out operations that are available through delegation to class. If these operations are shown, they should be marked as such. See illustration of possible realization in Fig. 2.20.

Rationale:

Priority: *medium*

RT16

A modelling tool that features dynamic typing should compensate for the lack of static typing. This could be achieved by adding information to code.

Rationale: If the representation of code is ambiguous, the analysis of large models can be painful and produce serious threats to system integrity.

Priority: *medium*

RT17

A modelling tool should support the generation of access methods according to adaptable policies.

Rationale: Generating access operations decreases development costs and contributes to comprehensibility and integrity of models and software.

Priority: *high*

RT18

A modelling tool should support the creation of user profiles. A user profile stores user's visualization preferences, which include font sizes, colour, the layout of window frames, etc.

Rationale: Adapting the representation of a model to individual preferences allows creating a more convenient and possibly more efficient environment. It is an unnecessary burden for users, if they have to specify their preferences each time they use a model (or an instance of the tool) that is used by others, too.

Priority: *medium*

RT19

With respect to keeping track of the evolution of a model, a tool should also store the users that were responsible for a certain modification.

Rationale: Sometimes, it is useful to find out who made certain modelling decisions, especially, if they seem unclear or inappropriate.

Priority: *medium*

RN1

The notation should be in line with widespread notations of conceptual modelling languages. That recommends to use shapes for the representation of classes similar to those used with the UML. This is especially the case for the representation of classes on M_1, because they are typically subject of traditional object models.

Rationale: The use of familiar representations is likely to promote the understandability of multi-level models.

Priority: *high*

RN2

The notation should build on that of the previous version of the FMML^x and account for critical feedback on the previous notation at the same time.

Rationale: The previous notation resulted from a long process of developing, using, and improving multi-level modelling languages. Therefore, it seems reasonable to assume that it is regarded as widely appropriate by those who used it. At the same time, certain aspects of the notation were subject of critical comments that should be accounted for.

Priority: *medium*

RN3

The additional concepts defined with the new version of the FMML^x , e.g., contingent-level classes, should be represented clearly.

Rationale: This requirement is a direct consequence of the general demand for understandability of models.

Priority: *high*

RN4

The representation of classification levels should follow certain design guidelines.

Rationale: In the previous version, the colours used for distinguishing classification levels were chosen in a widely arbitrary manner, which made it difficult to memorize what level a specific colour represented.

Priority: *medium*

2013

No 56 (July 2013)

Svensson, Richard Berntsson; Berry,Daniel M.; Daneva, Maya; Doerr, Joerg; Espana, Sergio; Herrmann, Andrea; Herzwurm, Georg; Hoffmann, Anne; Pena, Raul Mazo; Opdahl, Andreas L.; Pastor, Oscar; Pietsch,Wolfram; Salinesi, Camille; Schneider, Kurt; Seyff, Norbert; van de Weerd, Inge; Wieringa, Roel; Wnuk, Krzysztof (Eds.): "19th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2013). Proceedings of the REFSQ 2013 Workshops CreaRE, IWSPM, and RePriCo, the REFSQ 2013 Empirical Track (Empirical Live Experiment and Empirical Research Fair), the REFSQ 2013 Doctoral Symposium, and the REFSQ 2013 Poster Session""

No 55 (May 2013)

Daun, Marian; Focke, Markus; Holtmann, Jörg; Tenbergen, Bastian "Goal-Scenario-Oriented Requirements Engineering for Functional Decomposition with Bidirectional Transformation to Controlled Natural Language. Case Study "Body Control Module""

No 54 (March 2013)

Fischotter, Melanie; Goedicke, Michael; Kurt-Karaoglu, Filiz; Schwinning, Nils; Striewe, Michael "Erster Jahresbericht zum Projekt "Bildungsgerechtigkeit im Fokus" (Teilprojekt 1.2 – "Blended Learning") an der Fakultät für Wirtschaftswissenschaften"

2012

No 53 (December 2012)

Frank, Ulrich: "Thoughts on Classification / Instantiation and Generalisation / Specialisation"

No 52 (July 2012)

Berntsson-Svensson, Richard; Berry, Daniel; Daneva, Maya; Dörr, Jörg; Fricker, Samuel A; Herrmann, Andrea; Herzwurm, Georg; Kauppinen, Marjo; Madhavji, Nazim H; Mahaux, Martin; Paech, Barbara; Penzenstadler, Birgit; Pietsch, Wolfram; Salinesi, Camille; Schneider, Kurt; Seyff, Norbert; van de Weerd, Inge (Eds.): "18th International Working Conference on Requirements Engineering – Foundation for Software Quality. Proceedings of the Workshops RE4SuSy, REEW, CreaRE, RePriCo, IWSPM and the Conference Related Empirical Study, Empirical Fair and Doctoral Symposium"

No 51 (May 2012)

Frank, Ulrich: "Specialisation in Business Process Modelling – Motivation, Approaches and Limitations"

No 50 (March 2012)

Adelsberger, Heimo; Drechsler, Andreas; Herzig, Eric; Michaelis, Alexander; Schulz, Philipp ; Schütz, Stefan; Ulrich, Udo: "Qualitative und quantitative Analyse von SOA-Studien – Eine Metastudie zu serviceorientierten Architekturen"

2011

No 49 (December 2011)

Frank, Ulrich: "MEMO Organisation Modelling Language (2) – Focus on Business Processes"

No 48 (December 2011)

Frank, Ulrich: "MEMO Organisation Modelling Language (1) – Focus on Organisational Structure"

No 47 (December 2011)

Frank, Ulrich: "Multiperspective Enterprise Modelling – Requirements and Core Diagram Typs"

No 46 (December 2011)

Frank, Ulrich: "Multiperspective Enterprise Modelling – Background and Terminological Foundation"

No 45 (November 2011)

Frank, Ulrich; Strecker, Stefan; Heise, David; Kattenstroth, Heiko; Schauer, Carola: "Leitfaden zur Erstellung wissenschaftlicher Arbeiten in der Wirtschaftsinformatik"

No 44 (September 2011)

Berenbach, Brian; Daneva, Maya; Dörr, Jörg; Fricker, Samuel; Gervasi, Vincenzo; Glinz, Martin; Herrmann, Andrea; Krams, Benedikt; Madhavji, Nazim H; Paech, Barbara; Schockert, Sixten; Seyff, Norbert (Eds.): "17th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2011) – Proceedings of the REFSQ 2011 Workshops REEW, EPICAL and RePriCo, the REFSQ 2011 Empirical Track (Empirical Live Experiment and Empirical Research Fair), and the REFSQ 2011 Doctoral Symposium"

No 43 (February 2011)

Frank, Ulrich: "The MEMO Meta Modelling Language (MML) and Language Architecture. 2nd Edition"

2010

No 42 (December 2010)

Frank, Ulrich: "Outline of a Method for Designing Domain-Specific Modelling Languages"

No 41 (December 2010)

Adelsberger, Heimo; Drechsler, Andreas (Hrsg.): "Ausgewählte Aspekte des Cloud-Computing aus einer IT-Management-Perspektive – Cloud Governance, Cloud Security und Einsatz von Cloud Computing in jungen Unternehmen"

No 40 (October 2010)

Bürsner, Simone; Dörr, Jörg; Gehlert, Andreas; Herrmann, Andrea; Herzwurm, Georg; Janzen, Dirk; Merten, Thorsten; Pietschm, Wolfram; Schmid, Klaus; Schneider,Kurt; Thurimella, Anil Kumar: "16th International Working Conference on Requirements Engineering: Foundation for Software Quality – Proceedings of the Workshops CreaRE, PLREQ,RePriCo and RESC"

No 39 (May 2010)

Strecker, Stefan; Heise, David; Frank, Ulrich: "Entwurf einer Mentoring-Konzeption für den Studiengang M.Sc. Wirtschaftsinformatik an der Fakultät für Wirtschaftswissenschaften der Universität Duisburg-Essen"

No 38 (February 2010)

Schauer, Carola : "Wie praxisorientiert ist die Wirtschaftsinformatik? Einschätzungen von CIOs und WI-Professoren"

No 37 (January 2010)

Benavides, David; Batory, Don; Grunbacher, Paul (Eds.): "Fourth International Workshop on Variability Modelling of Software–intensive Systems"

2009

No 36 (December 2009)

Strecker, Stefan: "Ein Kommentar zur Diskussion um Begriff und Verstandnis der IT–Governance – Anregungen zu einer kritischen Reflexion"

No 35 (August 2009)

Rüngeler, Irene; Tüxen, Michael; Rathgeb, Erwin P.: "Considerations on Handling Link Errors in SCTP"

No 34 (June 2009)

Karastoyanova, Dimka; Kazhamiakan, Raman; Metzger, Andreas; Pistore, Marco (Eds.): "Workshop on Service Monitoring, Adaptation and Beyond"

No 33 (May 2009)

Adelsberger, Heimo; Drechsler, Andreas; Bruckmann, Tobias; Kalvelage, Peter; Kinne, Sophia; Pellinger, Jan; Rosenberger, Marcel; Trepper, Tobias: "Einsatz von Social Software in Unternehmen - Studie über Umfang und Zweck der Nutzung"

No 32 (April 2009)

Barth, Manfred; Gadatsch, Andreas; Kutz, Martin; Ruding, Otto; Schauer, Hanno; Strecker, Stefan: "Leitbild IT–Controller/–in . Beitrag der Fachgruppe IT–Controlling der Gesellschaft fur Informatik e. V."

No 31 (April 2009)

Frank, Ulrich; Strecker, Stefan: "Beyond ERP Systems: An Outline of Self–Referential Enterprise Systems – Requirements, Conceptual Foundation and Design Options"

No 30 (February 2009)

Schauer, Hanno; Wolff, Frank: "Kriterien guter Wissensarbeit - Ein Vorschlag aus dem Blickwienkel der Wissenschaftstheorie (Langfassung)"

No 29 (January 2009)

Benavides, David; Metzger, Andreas; Eisenecker, Ulrich (Eds.): "Third International Workshop on Variability Modelling of Software–intensive Systems"

2008

No 28 (December 2008)

Goedicke, Michael; Striewe, Michael; Balz, Moritz: "Computer Aided Assessments and Programming Exercises with JACK"

No 27 (December 2008)

Schauer, Carola: "Größe und Ausrichtung der Disziplin Wirtschaftsinformatik an Universitaten im deutschsprachigen Raum – Aktueller Status und Entwicklung seit 1992"

No 26 (September 2008)

Milen, Tilev; Bruno Muller–Clostermann:" CapSys: A Tool for Macroscopic Capacity Planning"

No 25 (August 2008)

Eicker, Stefan; Spies, Thorsten; Tschersich, Markus: "Einsatz von Multi–Touch beim Softwaredesign am Beispiel der CRC Card–Methode"

No 24 (August 2008)

Frank, Ulrich: "The MEMO Meta Modelling Language (MML) and Language Architecture - Revised Version"

No 23 (January 2008)

Sprenger, Jonas; Jung, Jürgen: "Enterprise Modelling in the Context of Manufacturing - Outline of an Approach Supporting Production Planning"

No 22 (January 2008)

Heymans, Patrick; Kang, Kyo–Chul; Metzger, Andreas, Pohl, Klaus (Eds.): "Second International Workshop on Variability Modelling of Software–intensive Systems."

2007

No 21 (September 2007)

Eicker, Stefan; Annett Nagel; Peter M. Schuler: "Flexibilität im Geschäftsprozessmanagement-Kreislauf"

No 20 (August 2007)

Blau, Holger; Eicker, Stefan; Spies, Thorsten: "Reifegradüberwachung von Software"

No 19 (June 2007)

Schauer, Carola: "Relevance and Success of IS Teaching and Research: An Analysis of the Relevance Debate"

No 18 (May 2007)

Schauer, Carola: "Rekonstruktion der historischen Entwicklung der Wirtschaftsinformatik: Schritte der Institutionalisierung, Diskussion zum Status, Rahmenempfehlungen für die Lehre"

No 17 (May 2007)

Schauer, Carola; Schmeing, Tobias: "Development of IS Teaching in North-America: An Analysis of Model Curricula"

No 16 (May 2007)

Müller-Clostermann, Bruno; Tilev, Milen: "Using G/G/m-Models for Multi-Server and Mainframe Capacity Planning"

No 15 (April 2007)

Heise, David; Schauer, Carola; Strecker, Stefan: "Informationsquellen für IT-Professionals - Analyse und Bewertung der Fachpresse aus Sicht der Wirtschaftsinformatik"

No 14 (March 2007)

Eicker, Stefan; Hegmanns, Christian; Malich, Stefan: "Auswahl von Bewertungsmethoden für Softwarearchitekturen"

No 13 (February 2007)

Eicker, Stefan; Spies, Thorsten; Kahl, Christian: "Softwarevisualisierung im Kontext serviceorientierter Architekturen"

No 12 (February 2007)

Brenner, Freimut: "Cumulative Measures of Absorbing Joint Markov Chains and an Application to Markovian Process Algebras"

No 11 (February 2007)

Kirchner, Lutz: "Entwurf einer Modellierungssprache zur Unterstützung der Aufgaben des IT Managements - Grundlagen, Anforderungen und Metamodell"

No 10 (February 2007)

Schauer, Carola; Strecker, Stefan: "Vergleichende Literaturstudie aktueller einführender Lehrbücher der Wirtschaftsinformatik: Bezugsrahmen und Auswertung"

No 9 (February 2007)

Strecker, Stefan; Kuckertz, Andreas; Pawlowski, Jan M.: "Überlegungen zur Qualifizierung des wissenschaftlichen Nachwuchses: Ein Diskussionsbeitrag zur (kumulativen) Habilitation"

No 8 (February 2007)

Frank, Ulrich; Strecker, Stefan; Koch, Stefan: "Open Model - Ein Vorschlag für ein Forschungspro-gramm der Wirtschaftsinformatik (Langfassung)"

2006

No 7 (December 2006)

Frank, Ulrich: "Towards a Pluralistic Conception of Research Methods in Information Systems Research"

No 6 (April 2006)

Frank, Ulrich: "Evaluation von Forschung und Lehre an Universitäten - Ein Diskussionsbeitrag"

No 5 (April 2006)

Jung, Jürgen: "Supply Chains in the Context of Resource Modelling"

No 4 (February 2006)

Lange, Carola: "Development and status of the Information Systems / Wirtschaftsinformatik discipline: An interpretive evaluation of interviews with renowned researchers, Part III - Results Wirtschaftsinformatik Discipline"

2005

No 3 (December 2005)

Lange, Carola: "Development and status of the Information Systems / Wirtschaftsinformatik discipline: An interpretive evaluation of interviews with renowned researchers, Part II - Results Information Systems Discipline"

No 2 (December 2005)

Lange, Carola: "Development and status of the Information Systems / Wirtschaftsinformatik discipline: An interpretive evaluation of interviews with renowned researchers, Part I - Research Objectives and Method"

No 1 (August 2005)

Lange, Carola: "Ein Bezugsrahmen zur Beschreibung von Forschungsgegenständen und -methoden in Wirtschaftsinformatik und Information Systems"

Research Group	Core Research Topics
Prof. Dr. F. Ahlemann Information Systems and Strategic Management	Strategic planning of IS, Enterprise Architecture Management, IT Vendor Management, Project Portfolio Management, IT Governance, Strategic IT Benchmarking
Prof. Dr. F. Beck Visualization Research Group	Information visualization, software visualization, visual analytics
Prof. Dr. T. Brinda Didactics of Informatics	Competence modelling and educational standards in Informatics, Students' conceptions in Informatics, Education in the digital world, Vocational education in Informatics
Prof. Dr. P. Chamoni MIS and Management Science / Operations Research	Information Systems and Operations Research, Business Intelligence, Data Warehousing
Prof. Dr.-Ing. L. Davi Research in Secure Software Systems	Software Security, Security of Smart Contracts, Trusted Computing, Hardware-assisted Security
Prof. Dr. K. Echtle Dependability of Computing Systems	Dependability of Computing Systems
Prof. Dr. S. Eicker Information Systems and Software Engineering	Process Models, Software-Architectures
Prof. Dr. U. Frank Information Systems and Enterprise Modelling	Enterprise Modelling, Enterprise Application Integration, IT Management, Knowledge Management
Prof. Dr. M. Goedicke Specification of Software Systems	Distributed Systems, Software Components, CSCW
Prof. Dr. V. Gruhn Software Engineering	Design of Software Processes, Software Architecture, Usability, Mobile Applications, Component-based and Generative Software Development
Prof. Dr. T. Kollmann E-Business and E-Entrepreneurship	E-Business and Information Management, E-Entrepreneurship/E-Venture, Virtual Marketplaces and Mobile Commerce, Online-Marketing
Prof. Dr. J. Marrón Networked Embedded Systems	Sensor Networks, Adaptive Systems, System Software for embedded systems, Data Management in mobile environments, Hoarding / Caching, Ubiquitous/Pervasive Computing, Semistructured databases
Prof. Dr. K. Pohl Software Systems Engineering	Requirements Engineering, Software Quality Assurance, Software-Architectures, Evaluation of COTS/Open Source-Components
Prof. Dr. Ing. E. Rathgeb Computer Network Technology	Computer Network Technology
Prof. Dr. S. Schneegaß Human Computer Interaction	Mobile, wearable, and ubiquitous computing systems, Implicit Feedback, Usable Security, Smart Clothing, Interaction in Virtual and Augmented Worlds, Ubiquitous Interaction
Prof. Dr. R. Schütte Business Informatics and Integrated Information Systems	Enterprise Systems, IS-Architectures, Digitalization of organisations, Information modelling, Scientific theory problems of the Business Informatics field
Prof. Dr. S. Stieglitz Professional Communication in Electronic Media / Social Media	Digital Enterprise / Digital Innovation, Digital Society

ISSN 1860-2770 (Print)
ISSN 1866-5101 (Online)