Ina Schaefer, Dimitris Karagiannis, Andreas Vogelsang (Hrsg.): Modellierung 2018,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2018 1

Extending a UML and OCL Tool for Meta-Modeling: Applications towards Model Quality Assessment

Khanh-Hoang Doan[1], Martin Gogolla[1]

Abstract:

For developing software in a model-driven style, meta- and multi-level modeling is currently gaining more and more attention. In this contribution, we propose an approach to extend a two-level modeling tool to three-level modeling by adding a meta-model at the topmost level. Standard OCL does not support reflective constraints, i.e., constraints concerning properties of the model like the depth of inheritance. By adding an auto-generated instance of the topmost level to the middle level, we can offer an option for writing reflective constraints and queries. We apply the extension to demonstrate the usefulness of meta-modeling for model querying and model quality assessment. A first proposal towards level-crossing constraints is also put forward.

Keywords: UML; OCL; Meta-modeling; Reflective constraints; Model querying; Model quality assessment.

1 Introduction

Within software development, Model-Driven Engineering (MDE) is playing now a more and more important role. MDE considers models as central development artifacts, for example by combining the UML (Unified Modeling Language) [Ob15b], and the OCL (Object Constraint Language) [CG12]. Meta-models play a crucial role in modeling as they define the structure of models, and meta-modeling [AK03, Bé05] and multi-level modeling [At14, At15, AGC16] has become a major research topic. Meta-modeling is closely connected to multi-level modeling because a UML and OCL model, which we call a *user model*, can be regarded an *instance model*, i.e., instantiation of a metamodel. The user model in turn may act as a *type model*, which may be instantiated again to a run-time instance model. Following this process, one obtains a modeling architecture with at least three levels.

Our starting point is a version of the tool USE (Uml-based Specification Environment) [GBR07, GH16] that supports two-level modeling. In this contribution we show how to extend the tool to three levels of modeling by adding the OMG (Object Management Group) UML meta-model to the topmost level. In order to make the work with different modeling levels easier, we provide the option to automatically take a simplified view on

[1] University of Bremen, Computer Science Department, E-Mail: {doankh,gogolla}@informatik.uni-bremen.de

the meta-model based on the elements of the current input user model. To offer access to the meta level, a meta-model instance corresponding to the user model is automatically generated and added to the middle level. By doing that, the user model turns into a instance model, and therefore modelers can write OCL constraints and queries about the model itself. We call this type of OCL expressions *reflective* constraints and queries. Thus the work in this paper supports meta-models and reflective OCL constraints and queries, i.e., expressions that treat the user model itself as data. Reflective querying helps to explore, understand and validate models, especially when we model large, complicated systems. For example, we can utilize reflective constraints to analyze a model and check for internal quality of a class diagram. In this paper, we also show the application of reflective constraints for *model quality* evaluation. The quality properties that we deal with in this paper are the problems on the class diagram might relate to general design issues, e.g., the naming of elements, or to questions regarding model metrics. Some similar approaches in this field have been presented, for example in the works [Ag11, AGO12, LGdL14]. Developers can use an appropriate method to evaluate their model and find drawbacks and problems in the model. The quality assessment method introduced in this contribution uses standard OCL in our three-layer modeling approach. Our three-layer modeling approach also offers an extension of standard OCL for level-crossing constraints, which are across all three levels, since the meta objects and run-time objects are accessible at the same time. An idea of extending OCL for level-crossing constraints is also presented in the later part of this paper.

The rest of the paper is organized as follows. Section 2 presents the general idea to extend our two-level tool to a three-level modeling tool. In Section 3 we show how to write reflective queries in the extended tool based on the available meta-model. The approach using reflective OCL constraints for quality assessment is introduced in Sect. 4. Section 5 presents a first proposal towards level-crossing constraints. The paper ends with some concluding remarks and future work in Section 7.

2 Tool-supported Meta Modeling

2.1 Meta-modeling in UML

The OMG has defined the Meta-Object-Facility (MOF) [Ob15a] as a fundamental standard for modeling. MOF provides a four-layer architecture for system modeling (three of them are shown in Fig. 1). Generally speaking, adjacent layers in the architecture are related by the instance-of relationship. This means that a lower layer is used for instantiating the next upper layer. One could also say that the same entities at a middle layer M_i can be (A) objects for the next upper layer M_{i+1} and (B) classes for the entities at the next lower layer M_{i-1}.

The top layer in the MOF architecture, named M3, is a meta-meta model. This meta-meta model is the language used to build the metamodels at the lower layer, called M2. The UML metamodel, which is used to describe the UML, is the most well-known example of a model

at M2 layer. The models at layer M2 describe the elements and the structure of the models at layer M1. Models at layer M1 can be, for instance, models written in UML. The last and bottom layer in the MOF architecture is the layer M0 (also called data layer). Models at this layer describe run-time instances (representations of real-world objects).

2.2 Basics of Meta Modeling in USE

In previous works, we have introduced USE (UML-based Specification Environment) [GBR07, GH16] as a two-level modeling tool, in which a user UML and OCL model at level M1 (class diagram and constraints) and run-time instances at layer M0 (object diagrams) are provided. In [Go15], ideas for experimenting with multi-level models in the two-level tool USE are presented. In the current contribution, we introduce an approach, in which the MOF architecture is integrated into USE for meta modeling. Roughly speaking, we now make the third OMG layer M2 explicitly available. Fig. 1 shows the general schema for our three-level modeling approach in the new version of USE. The model at layer M2 is the UML meta-model (the OMG superstructure) [Ob11]. This meta-model itself is an explicit UML class model formulated in USE with (currently) 63 classes and 99 associations. It is preloaded as a type model for all user UML models at layer M1 and is fixed during the modeling process. In the middle of our three-layer modeling approach there is a user UML and OCL model at layer M1, which is highlighted by the dashed rectangle in Fig. 1. The key point that makes our approach available for writing reflective constraints is an auto-generated meta-instance of the meta-model added to M1 level. Each meta-object is an instance of a meta-class and the number of generated meta-objects and links is based on the input user model. For example, if there are two classes in the user model, then two instances of meta class Class are generated. Our approach visits all user model elements (e.g., classes, attributes, operations, associations) and generates the corresponding meta-objects and links. Table 1 shows the mapping between the user model elements and the related meta-model classes and associations, which are the type elements for the generated meta-objects and links. In this work, we use the USE specific language SOIL (Simple OCL-based Imperative Language) [BG11] to create these meta-objects and links.

2.3 Three-layer Model Representation in USE

As discussed before, in the M1 layer we provide two views on the user model. The first view is a class diagram view, which can be seen as a type model for the object model at the lower layer M0. The second view is an object diagram view that is an instantiation of the UML meta-model at level M2 and that corresponds to the loaded UML user model. These two views represent the same information and are always in sync.

Fig. 2 contains an example of a three-level modeling representation in USE. The user model in this example is a simple model, with two classes, Employee and Department, and an

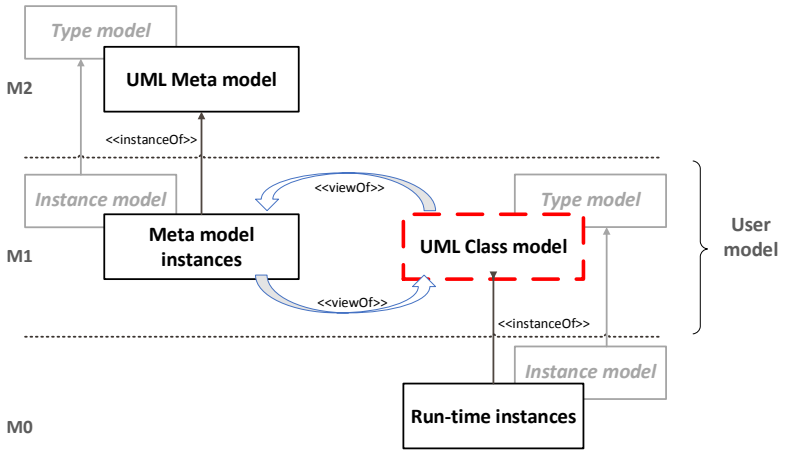

Fig. 1: General schema for three-layer model representation.

association WorksIn as shown in the right middle part of Fig. 2. As mentioned above, the full meta-model includes 63 classes and 99 associations. Therefore, viewing the full meta-model is not practical and sometimes not necessary, because many of the meta-model elements might not be used to describe the current user model. For example, the meta-model class Operation is not used to describe any element in the Employee-Department model. Starting from these observations, we provide a simplified view for the meta-model, as shown in the upper part of Fig. 2. To construct the simplified view from the full meta-model, we drop all unnecessary classes and associations, which are not needed for any element of the current user model. In the simplified view, we only show the meta-model elements, i.e., classes and associations, that the user model needs as type elements to instantiate model elements. Table 1 shows the mapping between the user model elements and the related meta-model classes and associations.

In each row, the item in the first column is a user model element, and the items in the second column are the classes that directly relate to the user model element, i.e., a typing model element. The third column contains the related meta-model associations (the subscript text includes the names of association ends corresponding to the classes). Based on this mapping, we can detect which meta-model elements will be displayed in the simplified view. For example, if a class in the user model contains attributes then the Property metaclass and two associations, i.e., $\text{Class}_{\text{class}} - \text{Property}_{\text{ownedAttribute}}$ and $\text{DataType}_{\text{DataType}} - \text{Property}_{\text{ownedAttribute}}$ will be shown. Concerning the example and as the result of the described mapping, only three classes, Class, Property, Association, and the corresponding associations are shown in the simplified meta-model view for the Department-Employee user model. Additionally, we still provide a full meta-model view, in case the developer wants to explore it.

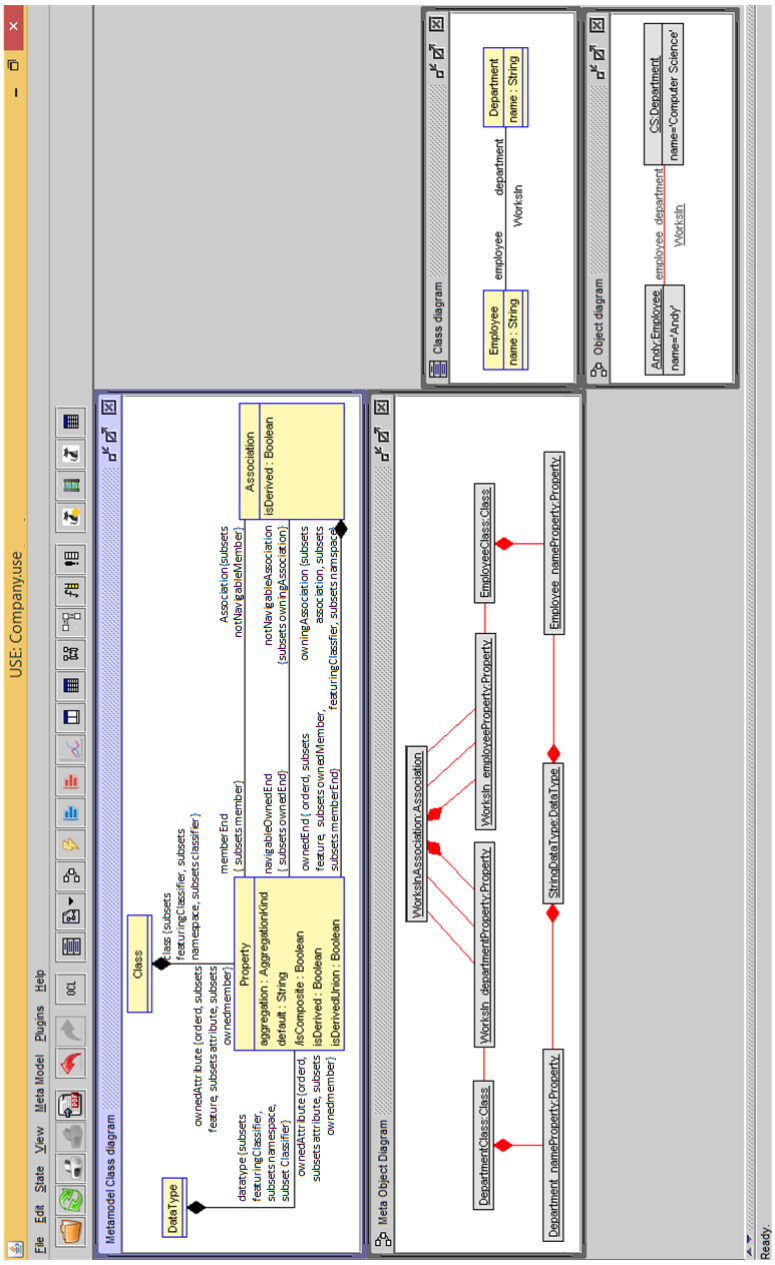

Fig. 2: Three-layer model representation in USE.

The left middle part in Fig. 2 is the user model represented as an instance of the meta-model. As can be seen, every element of the user model is an instance of a class from the meta model. Each instance is named as a combination of the name of the corresponding element from the user model and the meta-model class from which it is instantiated. For example, the object WorksInAssociation is an instance of metaclass Association and its name combines 'WorksIn', the name of the association from user model, and 'Association', the name of the meta-model class. The object diagram shown in the lower part of Fig. 2 is a run-time instance of the user model. It is the model at layer M0. 'CS' and 'Andy' are instances of the Department class and the Employee class, respectively.

There is a number of derived links between objects in the meta-instance view. However, to make the meta-instance view at layer M1 more focussing on the instanceOf aspect, we only show the direct links and do not show these derived links. The two views in layer M1 describe one model. They are equivalent and kept in sync. Each element in the user model class diagram view, e.g., a class, an attribute or an association, is presented as a meta-instance in the meta-instance view. If there is any change in the user model, e.g., a name change, an addition or a deletion of an element, the object diagram in meta-instance view will be updated. An example of a synchronous change on the views at layers M1 and M2 is presented and highlighted in Fig. 3. The change on the user model is made by

Tab. 1: Relationship between user model elements and meta-model elements.

User model elements	**Related UML meta-model classes**	**Related UML meta-model associations**
Class	Class	
Attribute	Property	$\text{Class}_{\text{class}} - \text{Property}_{\text{ownedAttribute}}$ $\text{DataType}_{\text{dataType}} - \text{Property}_{\text{ownedAttribute}}$
Association	Association	$\text{Class}_{\text{endType}} - \text{Association}_{\text{association}}$
Association End	Property	$\text{Association}_{\text{association}} - \text{Property}_{\text{memberEnd}}$ $\text{Association}_{\text{association}} - \text{Property}_{\text{navigableOwnedEnd}}$ $\text{Association}_{\text{association}} - \text{Property}_{\text{ownedEnd}}$
Operation	Operation	$\text{Class}_{\text{class}} - \text{Operation}_{\text{ownedOperation}}$ $\text{DataType}_{\text{dataType}} - \text{Operation}_{\text{ownedOperation}}$
Parameter	Parameter	$\text{Operation}_{\text{operation}} - \text{Parameter}_{\text{ownedParameter}}$
AssociationClass	AssociationClass	
Generalization	Generalization	$\text{Class}_{\text{class}} - \text{Class}_{\text{superClass}}$ $\text{Generalization}_{\text{Generalization}} - \text{Class}_{\text{specific}}$ $\text{Generalization}_{\text{Generalization}} - \text{Class}_{\text{general}}$
Redefined Attribute/ Redefined Association End		$\text{Property}_{\text{property}} - \text{Property}_{\text{redefinedProperty}}$
Subsetted Attribute/ Subsetted Association End		$\text{Property}_{\text{property}} - \text{Property}_{\text{subsettedProperty}}$

(A) adding the operation 'numberOfEmp(): Integer' into class Department. Consequently, two corresponding meta-instances are synchronously (B) added to the meta-instance view, i.e., the Department_numberOfEmp:Operation instance and the IntegerDataType:DataType instance for the return data type 'Integer'. And synchronously, the metaclass Operation and related associations are (C) added to the simplified meta-model view (at layer M2).

3 Tool-based Reflective Querying

The access to the meta-level supported by standard OCL [Ob06] is limited, therefore writing reflective queries, e.g., "find the classes related to a given class c and their relevant roles", is impossible. In this section, we will introduce how our approach supports more meta-level access capabilities for writing reflective queries within the extended tool.

3.1 Meta-level Accessibility in OCL

Standard OCL is a formal language for writing constraints and queries on UML models. OCL expressions are formulated on the level of classes (M1) and their semantics is applied on the level of objects (M0). Given a meta object t: OclType, the following table shows the list of supported OCL meta-level access capabilities.

Tab. 2: OCL built-in meta-level access

Expression	Semantics
t.name() : String	Get the name of the type t
t.attributes() : Set(String)	Get the set of names of all attributes of t
t.operations() : Set(String)	Get the set of names of all operations of t
t.associationEnds() : Set(String)	Get the set of names of all association ends navigable from t
t.supertypes() : Set(OclType)	Get the set of all direct supertypes of t
t.allSupertypes() : Set(OclType)	Get the transitive closure of the set of all supertypes of t

As we can see, with these limited meta-level access capabilities, standard OCL cannot express a number of reflective queries and constraints. The following list presents several reflective queries that cannot be expressed with the standard OCL.

1. Find all classes related to a given class
2. Find names of all subclasses of a given class
3. Find all abstract classes
4. Find all classes that have more than 10 attributes

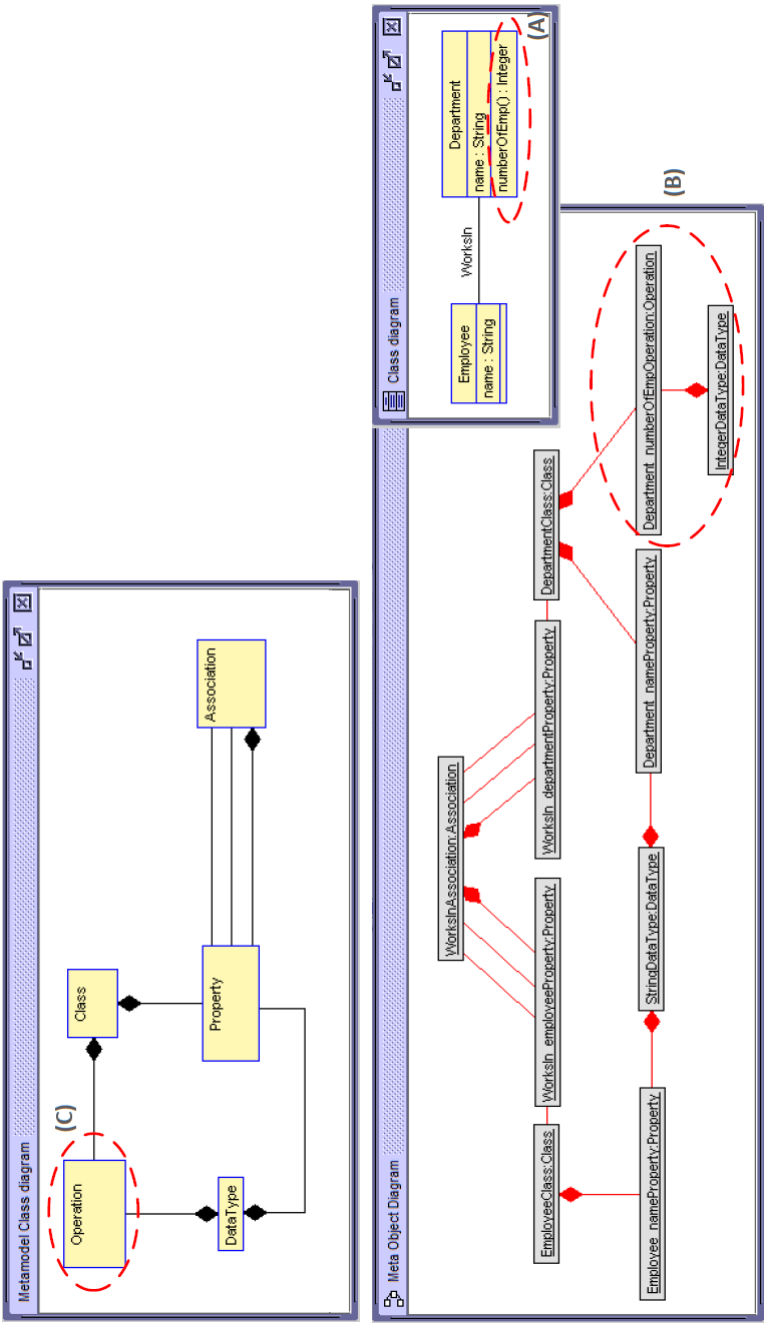

Fig. 3: Synchronous changes on the views of layer M1 and M2.

5. Find all classes that have more than 5 subclasses
6. Calculate the number of classes in a user model
7. Check for the setter and getter methods of all attributes
8. Find all classes of a user model that have no subclass

These queries, however, can be expressed with our three-level modeling approach introduced in the previous section. In the next section, we will show how to formulate and execute reflective queries in the extended tool.

3.2 Writing Reflective OCL Querying in Tool USE

As introduced in the previous section, our approach supports a three-layer UML and OCL specification: instances, model, and meta-model. Through the tool support, one can access the meta-model and create OCL queries for the user model by considering it as an instance of the meta-model. Model querying using the meta-model approach provides possibilities for considering the elements contained in a model, for example, by accessing the attributes, operations, and referenced elements of a given model element, by executing comprehensions and quantified expressions. A query is an OCL expression on the meta-model layer, and the result is a Boolean value or a set of user model elements in form of instances of a meta-model type element. Model queries such as "find the classes related to a class and their relevant roles" cannot be formulated in OCL directly. However, our meta-modeling approach can deal with this kind of model query. For example, the query "find classes related to class Department via an association" on the Employee-Department example can be formulated by the following OCL expression and executed by our tool as shown in Fig. 4.

The Association meta class is the type element for associations in the user model and endType is an end of a derived association that can be used to navigate from the Association meta class to the Class metaclass. DepartmentClass is the meta-instance of the Class metaclass; its name is a combination of the user model element and the corresponding metaclass. The result of executing this query, i.e., Bag{EmployeeClass}, is also shown in Fig. 4.

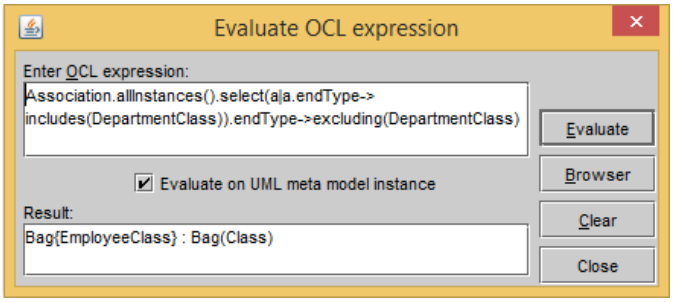

Fig. 4: Model query example: Find related classes.

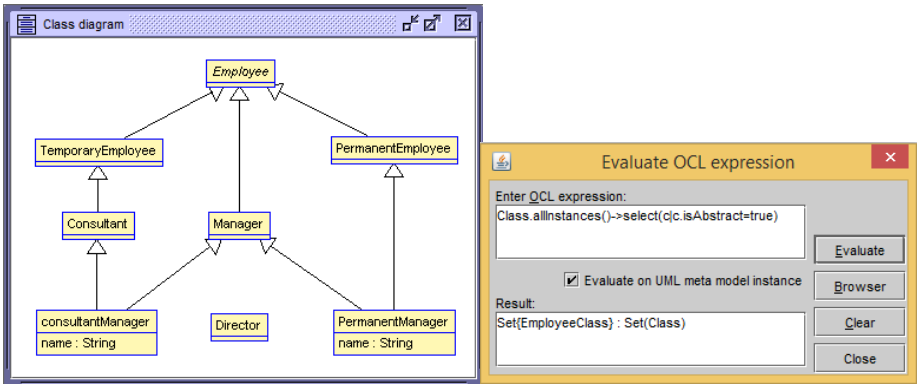

Fig. 5: Model query example: Find abstract classes.

Another example for model querying is presented in Fig. 5. The example there is an Employee hierarchy model, a typical subclass-superclass generalization model with a four-level inheritance structure. For example, one might want to find all abstract classes within this model. The OCL query to perform this task is stated in Fig. 5. In particular, isAbstract is a Boolean attribute of the metaclass Class in order to define whether a class at level M1 (in a user model) is abstract or not.

4 Model Quality Assessment with Reflective Constraints

Writing reflective constraints is now possible with our meta-modeling approach. Reflective constraints can be exploited for many applications, one of them is model quality assessment. Model quality assessment helps modelers to detect errors or mistakes on their models, to fix bugs and to improve the models. These assessment properties might include design properties: absence of isolated classes, respecting naming conventions (e.g., the name of every element must obey the camelCase convention) or metrics properties (e.g., a generalization hierarchy is not too deep). By visual inspection, we can identify several quality problems in the example model in Fig. 5:

1. There is one isolated class, i.e., Director. An isolated class is a class which is not involved in an association or in the inheritance hierarchy.
2. The name of the class consultantManager does not start with a capital letter (assuming the class names should obey the camelCase convention).
3. The attribute name : String is repeated in all subclasses of the class Manager. It should be defined in the superclass.

However, evaluating and detecting these kinds of quality problems on large and complicated models might take time and might even be impractical. In this section, we introduce a

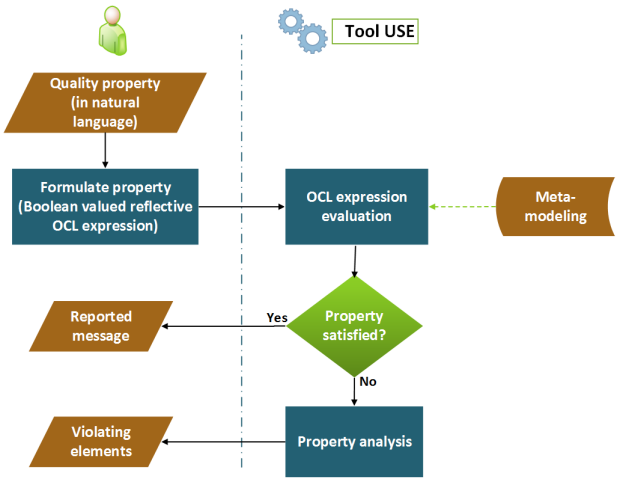

Fig. 6: The workflow of model quality assessment process

proposal that employs OCL utilizing the meta-level modeling approach as presented before. Thus, we can automatically evaluate quality properties of a user model.

Fig 6 shows the workflow of the model quality evaluation process. Firstly, the property must be formulated by the developer as a Boolean-valued reflective OCL expression. The next evaluation and analysis steps will be performed by tool USE. The reflective OCL expression will be evaluated. If the evaluation yields True, the property is satisfied and the model respects this property. On the contrary, if the property fails, the developer might be interested in the parts of the model that violate the property. Returning to the example in Fig. 5, we want to check the first problem mentioned in the section beginning, i.e., whether there are isolated classes in the user model. To achieve this, we formulate an OCL expression for the property.

```
Class.allInstances()->select(c | c.typeElement->isEmpty() and
  c.superClass->isEmpty() and c.subClass->isEmpty())->isEmpty())
```

In this example, we navigate from a metaclass c to related associations through the typeElement role name. The superClass and subClass role names are used to navigate from the metaclass c to its superclass and subclass, respectively. The assessment result is False as shown in Fig. 7. That means the property is violated. It can be seen from the user class diagram in Fig. 5 that there is one isolated class, i.e., the class Director.

In the case of simple models, one can manually figure out the elements that cause the violation of an assumed property. However, with a large, complicated model, this can be hard work and can sometimes be impossible. Our tool supports designers to analyze such properties and to look for the reason for the unsatisfiability. Particularly the USE evaluation

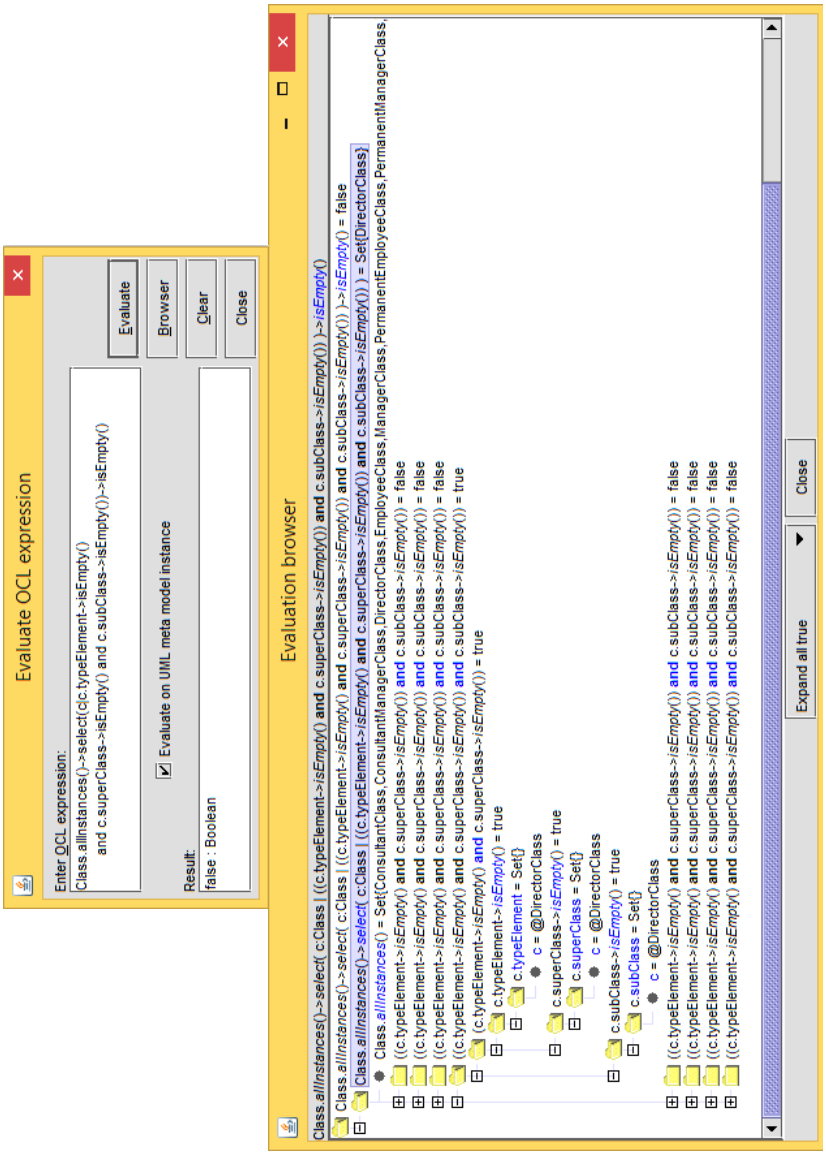

Fig. 7: Example of model assessment and analysis.

browser allows developers to dive into the details of the formula evaluation and identify the spots in the object diagram that contribute to the fact that the formula is not satisfied.

The USE evaluation browser in the lower part of Fig. 7 can be obtained by clicking the 'Browser' button on the right hand side of the OCL expression evaluation window in Fig. 7. The browser window decomposes the expression into sub-parts in a hierarchical structure, and every part is evaluated. From the evaluation browser, we can see that there is only one violating element, i.e., one isolated class, which is the Director class. For further analysis, one can expand sub-expressions and explore the evaluation of other sub-parts of the formula as shown in Fig .7.

5 Towards an Approach for Level-Crossing Constraints

Level-crossing plays an important role in multi-level modeling. Standard OCL, however, only supports formulating constraints on the level of classes (M1) and their semantics concerns the level of objects (M0). That means it is impossible to write expressions on objects at different linguistic levels, e.g., the expression "Class.allInstances().allInstances()" is syntactically invalid in OCL. Looking back to our approach on three-level modeling, we can see that developers now can access meta objects as well as run-time objects at the same time. Therefore, we can say that our three-level modeling approach offers enough semantics for extending OCL for multi-level-crossing constraints. Let us consider the following level-crossing constraint, which is an invalid, ill-typed constraint in standard OCL.

```
Class.allInstances().allInstances()->forAll(age>18)
```

Assuming every class in the model has an age attribute with Integer type, this constraint ensures that the value of the age attribute of all instances of all classes is over 18. As we can see, the semantics of the first part of the constraint, i.e., Class.allInstances(), concerns the meta-level, and the semantics of the second part of the expression, i.e., allInstances()->forAll(age > 18), concerns the model level. The result of the meta-level part has type Set(Class). Unfortunately, the allInstances() operation (in standard OCL [Ob06]) is applicable only on type Class.

To overcome this issue, one could work with the forAll collection operation between the meta-level expression and the model-level expression. If we handle a term of type Set(Class) with the forAll iterator this gives the option to access a single Class, on which the model-level expression can be applied.

```
Class.allInstances()->forAll(c | # c.allInstances()->forAll(age > 18) # )
```

Naturally, we have to distinguish meta-level sub-expressions and model-level sub-expressions. In other words, we have to indicate in OCL, which sub-expression belongs to which level (meta-level or model-level). To achieve this, we introduce an additional notation in OCL expressions, i.e., #...#. This indicates that the expression within #...# is a model-level

expression. As the result, we propose a formula template for a level-crossing constraint as shown below.

<meta-level OCL expression>->forAll(c | # <model-level OCL expression> #)

Generally speaking, using this formula template, one can write constraints that go from the M2 to the M0 level through the M1 level. This capability supports writing more powerful and flexible constraints. Instead of using the universal quantification it would also be possible to use an existential quantification. Other OCL collection operations, e.g. one, are feasible as well. Working out details is subject to future work. Our proposed level-crossing formula template has a few restrictions: (a) the meta-level expression must return the type Set(Class), (b) the model-level expression must be a Boolean-valued expression, and (c) the result of the overall level-crossing expression is always a Boolean value.

6 Related Work

There is a number of other proposals, which are related and similar to our work, that have been introduced in recent years. The tool Melanee [AG12] is designed as an Eclipse plug-in, supports strict multi-level metamodeling and facilitates general purpose languages as well as domain specific languages. Another tool is MetaDepth [dLG10] allowing linguistic as well as ontological instantiation with an arbitrary number of meta-levels supporting the potency concept. The framework Modelverse introduced in [Mi14] can be used to model a four-level language hierarchy. The work in [BKK16, IGS14] uses F-Logic as an implementation basis for multi-level models including constraints. In contrast to these approaches, our contribution deals with traditional two-level UML/OCL modeling approaches by extending them for meta-modeling and exploits added meta-data for writing reflective constraints and level-crossing constraints with OCL. One commonality between our work and these above mentioned approaches is the introduction of elements in the middle level that have both type and instance facets.

The idea of copying the M2-model instance to lower levels in the MOF meta-model architecture and exploiting it for reflective constraint writing is also presented in [Dr16]. In that paper, the meta instance is added to the M0 level, together with the run-time instances, through instantiation and reification processes. Adding elements to the M0 level is a major difference between the work in [Dr16] and our approach, because in our work, the meta instance is generated and added to the M1 level. Therefore, in order to write a reflective constraint or query with our approach, we only need to go from the M2 level to the M1 level. This means we do not need to extend the OCL for writing reflective constraints or queries.

7 Conclusion

This contribution has proposed an extension of the tool USE that supports three-level modeling where the middle level can be seen at the same time as an object diagram, i.e.,

the instantiation of the upper level model, and as a class diagram, i.e., the type model for the lower level. Based on these ideas, we present an approach for reflective constraints and queries within the extended tool and the application of this approach to model quality assessment. A first proposal towards level-crossing constraints was also introduced: a proposal that offers writing more powerful and flexible constraints.

Future work includes the following topics. First of all, we would like to work out within our approach formal definitions for notions like potency or strictness. Developer support for these notions should then be explored. The user interface in our tool USE for model querying and quality evaluation can be strengthened as well. For instance, one option might be to highlight the result of meta-level queries in the user class diagram. Another open item would be to implement a library of pre-defined quality assessment properties. With the integration of three-level modeling in the tool USE, more work on model metrics seems to be a promising direction for a USE extension. The proposal for level-crossing constraints must be implemented and extended to cover other formula templates. Last but not least, complex examples and case studies, especially case studies from large applications, must check the practicability of the proposal.

References

[Ag11] Aguilera, David; García-Ranea, Raúl; Gómez, Cristina; Olivé, Antoni: An Eclipse Plugin for Validating Names in UML Conceptual Schemas. In: Proc. ER 2011 Workshops FP-UML, MoRE-BI, Onto-CoM, SeCoGIS, Variability@ER, WISM. pp. 323–327, 2011.

[AG12] Atkinson, Colin; Gerbig, Ralph: Melanie: Multi-level Modeling and Ontology Engineering Environment. In: Proc. 2nd Int. Master Class MDE: Modeling Wizards, co-located with MODELS 2012. MW'12, pp. 7:1–7:2, 2012.

[AGC16] Atkinson, Colin; Grossmann, Georg; Clark, Tony, eds. Proc. 3rd Int. Workshop Multi-Level Modelling co-located with MoDELS 2016, volume 1722 of CEUR Workshop Proceedings. CEUR-WS.org, 2016.

[AGO12] Aguilera, David; Gómez, Cristina; Olivé, Antoni: A Method for the Definition and Treatment of Conceptual Schema Quality Issues. In: Proc. 31st Int. Conf. ER 2012. pp. 501–514, 2012.

[AK03] Atkinson, C.; Kuhne, T.: Model-Driven Development: A Metamodeling Foundation. IEEE Software, 20(5):36–41, 2003.

[At14] Atkinson, Colin; Grossmann, Georg; Kühne, Thomas; de Lara, Juan, eds. Proc. 1st Int. Workshop Multi-Level Modelling co-located with MoDELS 2014, volume 1286 of CEUR Workshop Proceedings. CEUR-WS.org, 2014.

[At15] Atkinson, Colin; Grossmann, Georg; Kühne, Thomas; de Lara, Juan, eds. Proc. 2nd Int. Workshop Multi-Level Modelling co-located with MoDELS 2015, volume 1505 of CEUR Workshop Proceedings. CEUR-WS.org, 2015.

[Bé05] Bézivin, Jean: On the Unification Power of Models. Software & Systems Modeling, 4(2):171–188, 2005.

[BG11] Büttner, Fabian; Gogolla, Martin: Modular Embedding of the Object Constraint Language into a Programming Language. In: Formal Methods, Foundations and Applications: 14th Brazilian Symposium. Springer Berlin Heidelberg, pp. 124–139, 2011.

[BKK16] Balaban, Mira; Khitron, Igal; Kifer, Michael: Multilevel Modeling and Reasoning with FOML. In: IEEE Int. Conf. SWSTE. pp. 61–70, 2016.

[CG12] Cabot, Jordi; Gogolla, Martin: Object Constraint Language (OCL): A Definitive Guide. In (Bernardo, Marco; Cortellessa, Vittorio; Pierantonio, Alfonso, eds): Formal Methods for Model-Driven Engineering, LNCS 7320, pp. 58–90. Springer, 2012.

[dLG10] de Lara, Juan; Guerra, Esther: Deep Meta-modelling with MetaDepth. In: Proc. 48th Int. Conf. TOOLS 2010. pp. 1–20, 2010.

[Dr16] Draheim, Dirk: Reflective Constraint Writing. In: Special Issue on Database- and Expert-Systems Applications on Transactions on Large-Scale Data- and Knowledge-Centered Systems XXIV - Volume 9510. Springer-Verlag New York, Inc., pp. 1–60, 2016.

[GBR07] Gogolla, Martin; Büttner, Fabian; Richters, Mark: USE: A UML-based Specification Environment for Validating UML and OCL. Sci. Comput. Program., 69(1-3):27–34, 2007.

[GH16] Gogolla, Martin; Hilken, Frank: Model Validation and Verification Options in a Contemporary UML and OCL Analysis Tool. In (Oberweis, Andreas; Reussner, Ralf, eds): Proc. Modellierung (MODELLIERUNG'2016). GI, LNI 254, pp. 203–218, 2016.

[Go15] Gogolla, Martin: Experimenting with Multi-Level Models in a Two-Level Modeling Tool. In: Proc. 2nd Int. Workshop Multi-Level Modelling co-located with MoDELS 2015. pp. 3–12, 2015.

[IGS14] Igamberdiev, Muzaffar; Grossmann, Georg; Stumptner, Markus: An Implementation of Multi-Level Modelling in F-Logic. In: Proc. Workshop Multi-Level Modelling co-located with MoDELS 2014. pp. 33–42, 2014.

[LGdL14] López-Fernández, Jesús J.; Guerra, Esther; de Lara, Juan: Assessing the Quality of Meta-Models. In: Proc. 11th Workshop MoDeVVa@MODELS 2014. pp. 3–12, 2014.

[Mi14] Mierlo, Simon Van; Barroca, Bruno; Vangheluwe, Hans; Syriani, Eugene; Kühne, Thomas: Multi-Level Modelling in the Modelverse. In: Proc. Workshop Multi-Level Modelling co-located with MoDELS 2014. pp. 83–92, 2014.

[Ob06] Object Management Group – OMG: . OMG: Object Constraint Language, version 2.0, 2006.

[Ob11] Object Management Group – OMG: . OMG Unified Modeling Language(OMG UML), Superstructure, version 2.4.1, 2011.

[Ob15a] Object Management Group – OMG: . OMG Meta Object Facility (MOF) Core Specification, version 2.5, 2015.

[Ob15b] Object Management Group – OMG: . Unified Modeling Language Specification, version 2.5, 2015.