Endogenous Metamodeling Semantics for Structural UML 2 Concepts

Lars Hamann and Martin Gogolla

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany
{lhamann,gogolla}@informatik.uni-bremen.de
http://www.db.informatik.uni-bremen.de

Abstract. A lot of work has been done in order to put the Unified Modeling Language (UML) on a formal basis by translating concepts into various formal languages, e.g., set theory or graph transformation. While the abstract UML syntax is defined by using an endogenous approach, i. e., UML describes its abstract syntax using UML, this approach is rarely used for its semantics. This paper shows how to apply an endogenous approach called metamodeling semantics for central parts of the UML standard. To this end, we enrich existing UML language elements with constraints specified in the Object Constraint Language (OCL) in order to describe a semantic domain model. The UML specification explicitly states that complete runtime semantics is not included in the standard because it would be a major amount of work. However, we believe that certain central concepts, like the ones used in the UML standard and in particular property features as subsets, union and derived, need to be explicitly modeled to enforce a common understanding. Using such an endogenous approach enables the validation and verification of the UML standard by using off-the-shelf UML and OCL tools.

Keywords: Metamodeling, Semantics, Validation, UML, OCL.

1 Introduction

In order to describe the abstract syntax of modeling languages, well-known concepts like classes, associations, and inheritance are used to express the structure of a language. These elements are commonly used in combination with a textual language to express further well-formedness rules which cannot be expressed using a graphical syntax. To improve the expressiveness of graphical modeling languages, especially when using complex inheritance relations, additional annotations have been developed to express more detailed information about the relation between model elements. Examples of these annotations are the subsets relations between properties and tagging a property as a derived union. The abstract syntax definition of the UML [23,26] uses these newer modeling elements

A. Moreira et al. (Eds.): MODELS 2013, LNCS 8107, pp. 488–504, 2013.

since UML 2. Such a distinguished usage calls for the need of a precise definition at the syntax level (design time) and also on the semantic level (runtime)[1].

In this paper, we present an endogenous approach to specify the syntax and the semantics of central concepts of modeling languages. To this end, we use the same formalism, i. e., class diagrams enriched with constraints expressed in the Object Constraint Language (OCL) [24,32], as used currently for the syntax description of modeling languages. To demonstrate our approach we choose particular UML language features (subsets, union and derived), but the same method may be applied to all UML language elements. The language features we choose are also important on their own, because they are used in MOF (i. e. as a description language for UML) without having a proper formal semantics currently. Our work is different to other approaches, like for example [1,19], that define a formal semantics for the modeling elements mentioned above, in the sense, that we use the same languages to describe the syntax and the semantics instead of translating syntactical elements into a different formalism.

The rest of this work is structured as follows: In the next section we describe the concept of metamodeling semantics. In Sect. 3 we explain our approach for metamodeling the runtime semantics of modeling elements by using well-known examples. Section 4 identifies benefits arising when using tool-based validation of modeling concepts. Before the paper ends with a conclusion and future work, we discuss related approaches in Sect. 5.

2 Metamodeling Semantics

The notion *Metamodeling Semantics* can be explained well by quoting a statement from [16]:

> Metamodeling semantics is a way to describe semantics that is similar to the way in which popular languages like UML are defined. In metamodeling semantics, not only the abstract syntax of the language, but also the semantic domain, is specified using a model.

Metamodeling a language by defining the abstract syntax using a graphical modeling language combined with a formal textual language to express well-formedness rules is a well-known technique. The UML specification for example uses UML (or MOF which itself uses UML) in combination with the Object Constraint Language (OCL) to define its abstract syntax. In [16] this is called the *Abstract Syntax Model (ASM)*, which defines the valid structures in a model. The same technique is rarely used to define the semantics of a language, i. e., to specify a *Semantic Domain Model (SDM)* of a modeling language. A *semantic domain* defines the meaning of a particular language feature, whereas a semantic domain model describes this meaning by modeling the runtime behavior of a (syntactically) valid model using its runtime values and applying meaning to

[1] In this work, we distinguish between design time and runtime by using classes and objects. Note, that this distinction is not always appropriate.

them. For example, later we will see that in the UML there is the class `Class` in the abstract syntax part, and there is the class `InstancesSpecification` in the semantic domain part which together can describe (through an appropriate association) that a class (introduced at design time) is interpreted (at runtime) by a set of objects, formally captured as instance specifications. Another publicly available example for metamodeling semantics can be found in Section 10 of the OCL specification [24]. It defines constraints on values, i. e., runtime instances, which are part of the SDM. For example, the runtime value of a set is constrained as follows:

```
context SetTypeValue inv: self.element->isUnique(e : Element | e.value)
```

The central idea behind the approach in [24] is to describe the runtime behavior of OCL using OCL, which is similar to the UML metamodel described by UML models. While this is done in the UML to constrain the metamodel level M1, i. e., the valid structure of models, very little formal information is given for the level M0. Nearly only, the structure for the runtime snapshots is specified, but little use is made of defining runtime constraints in a formal language like OCL. An excerpt of the UML metamodel which shows important elements for our work is shown in Fig. 1. The diagram combines elements from roughly six syntax diagrams of the UML metamodel. On the left side, the ASM (syntax) of the UML is shown. On the right, the SDM (semantics) elements are given as they are present in the current specification. In the next section we define runtime constraints on the semantic domain model for several modeling constructs which are frequently used in the definition of the UML metamodel, but are only defined in an informal way with verbal descriptions in the current UML.

3 OCL-Based Instance and Value Semantics

In this section we describe our approach of metamodeling semantics for different language features. We start with commonly used constraints on properties and how they can be described without leaving the technology space. Next we explain the semantics for evaluating derived properties.

3.1 Subsetting and Derived Unions

We explain our proposal by starting with a basic class diagram, which uses subsetting and union constraints on attributes of classes. Later on, we extend this diagram by using subsetting and union on associations. Subsetting and union constraints on properties (a property can be an attribute or an association end) define a relation between these two properties. The values of a subsetting property must be a subset of the values for the subsetted property. Union can be used on a single property. Its usage defines that the values of a property are the union of all its subsetting properties.

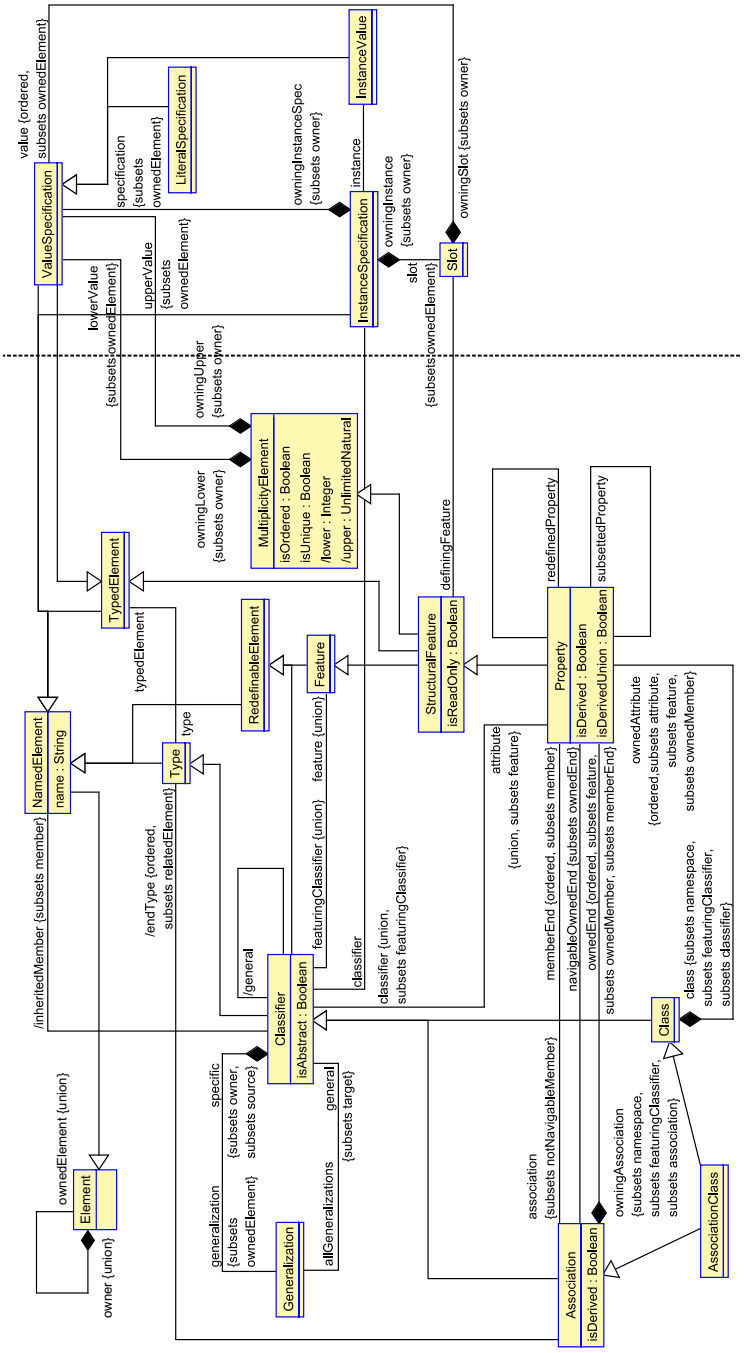

Fig. 1. Combined view of UML metamodel elements important for our work

Figure 2 shows a simple model of vehicles (c. f. [4]). A vehicle consists of vehicle parts. For a car, information about the front and back wheels is added to the class Car. Because these wheels are part of the overall vehicle, the properties `front` and `back` are marked as subsets of the general property `part`. The property `part` itself is marked as a derived `union` of all of its subsets. Furthermore, the subsetting properties restrict the lower and upper bounds of the wheels to the common number of wheels for a car (2 is equivalent to 2..2). A valid object

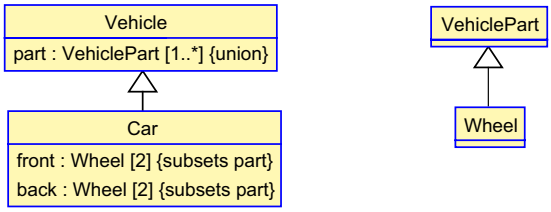

Fig. 2. Class diagram using `subsets` and `union` on attributes

diagram w. r. t. the given class diagram is shown in Fig 3. For this simple diagram, one can see directly that the intended constraints are fulfilled. However, for more complicated models, an automatic validation is required. If the used modeling language would not provide `subsets` and `union` constraints, a modeler could still specify constraints on the classes `Vehicle` and `Car`:

```
context Vehicle inv partIsUnion: let selfCar = self.oclAsType(Car) in
    selfCar <> null implies self.part = selfCar.front->union(selfCar.back)
context Car inv frontIsSubset: self.part->includesAll(self.front)
context Car inv backIsSubset:  self.part->includesAll(self.back)
```

However, these constraints would strongly couple the abstract class `Vehicle` and its subclass `Car`, because `Vehicle` needs information about its subclasses to validate the union constraint. This breaks well-known design guidelines. The above constraints are similar to the generated constraints from [20]. Using such an automatic approach would reduce the coupling.

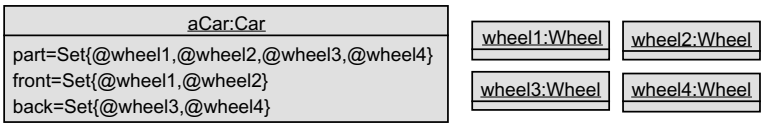

Fig. 3. A valid object diagram of the class diagram shown in Fig. 2

To allow a generic usage of these constraints the UML provides the ability to specify subset relations between properties using a reflexive association on

`Property` (which represents class attributes and association ends) and to mark a property as a derived union (see Fig. 1). Further, several well-formedness OCL rules are given, to ensure the syntactical correctness of the usage. For example, the type of the subsetting property must conform to the type of the subsetted end [23, p. 126]. However, information about the semantics of the UML language element `subsets` is only provided textually, not in a formal way. We propose to add (what we call) runtime semantics by means of OCL constraints to the already present elements describing runtime elements. For the above example, a constraint describing the runtime semantics of subsets can be specified on the UML metaclass `Slot` (a slot allows, for example, to assign an attribute value to an attribute):

```
context Slot inv subsettingIsValid:
let prop = self.definingFeature.oclAsType(Property) in
(prop <> null and prop.owner.oclIsKindOf(Class)) implies
 prop.subsettedProperty->forAll(subsettedProp |
  let subsettedValues = self.owningInstance.slot->
   any(definingFeature=subsettedProp).value.getValue()->asSet() in
  let currentValues = self.value.getValue()->asSet() in
  subsettedValues->includesAll(currentValues))
```

This constraint checks for each slot that defines a value or values for an attribute of a class, if it is a subset of the values defined by the slots of the subsetted properties. Because this constraint only considers attributes of classes, the navigation to the slots of the owning instance of the context slot is enough. For associations, and especially for associations with more than two ends, the calculation of the values to be considered is more complicated.

A class diagram which makes use of `subsets` and `union` on association ends is given in Fig. 4. The previously specified attributes `part` and `front` are changed to association ends, while the attribute `back` is left out in order to keep the following examples at a moderate size.

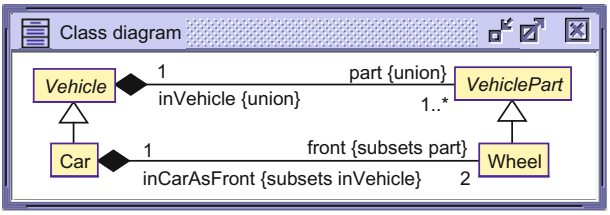

Fig. 4. Class diagram using `subsets` and `union` on association ends

Figure 5 shows an example instantiation of the class diagram. The links shown as a solid line are inserted by the user, while the dashed links are automatically calculated by our tool, because they are part of a derived union. In our tool, all

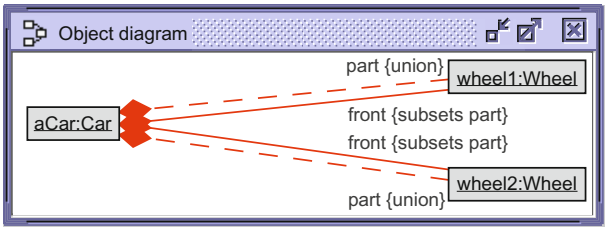

Fig. 5. A valid object diagram of the class diagram shown in Fig. 4

derived links (either established through a derived union or through an explicit derived association end) are shown as dashed links.

The object diagram in Fig. 6 shows an instantiation of the UML metamodel representing the class diagram of Fig. 4 at the top and the object diagram shown in Fig. 5 at the bottom. This figure intentionally includes so many dashed lines and compositions, in order to show the inherent complexity of the UML metamodel. This complexity can automatically be revealed by using our tool. In Sect. 4 we are going to explain these so-called virtual links in more detail. On the other side, these virtual links allow us to suppress certain elements in the object diagram to make it easier to be read. For example, the generalization relationships are only shown as derived links between the classes leaving out the generalization instance. To be more concrete, in the left upper part of Fig. 6 the dashed link between `Class3` (Vehicle) and `Class4` (Car) corresponds to the left generalization arrow in Fig. 4. We use this diagram in the following to explain an extended runtime semantics which also covers associations.

A runtime semantics for subsetting that covers attributes and association ends must consider all tuples of instances which are linked to a subsetted property and the set of instances linked to this tuple at the subsetting end. For the previously shown example on attributes, this tuple contains only one element, namely the defining instance, whereas for association ends of an association with n ends, this tuple contains $n-1$ elements. We accomplish this by using a query operation called `getConnectedObjects()` which is similar to the operation `Extent::linkedObjects(...)` defined in the MOF specification[22], but covers n-ary associations, properties, and derived unions. We do not show the operation in detail, because it is rather lengthy[2]. The query operation uses the metaclasses of the semantic domain model to obtain all connections specified for a property. For this, it navigates to all instance specifications to consider and their owned slots. If a property is defined as a derived union, this operation is recursively invoked on all properties subsetting the derived union property and collects all connected values in a single set, i. e., it builds the union of the values. To give a more detailed view of the usage of this central operation, Fig. 7 shows the result of invoking it on the property `part` using the state shown in Fig. 6.

[2] Interested readers are referred to the USE distribution which contains a well-defined subset of the UML metamodel including this operation.

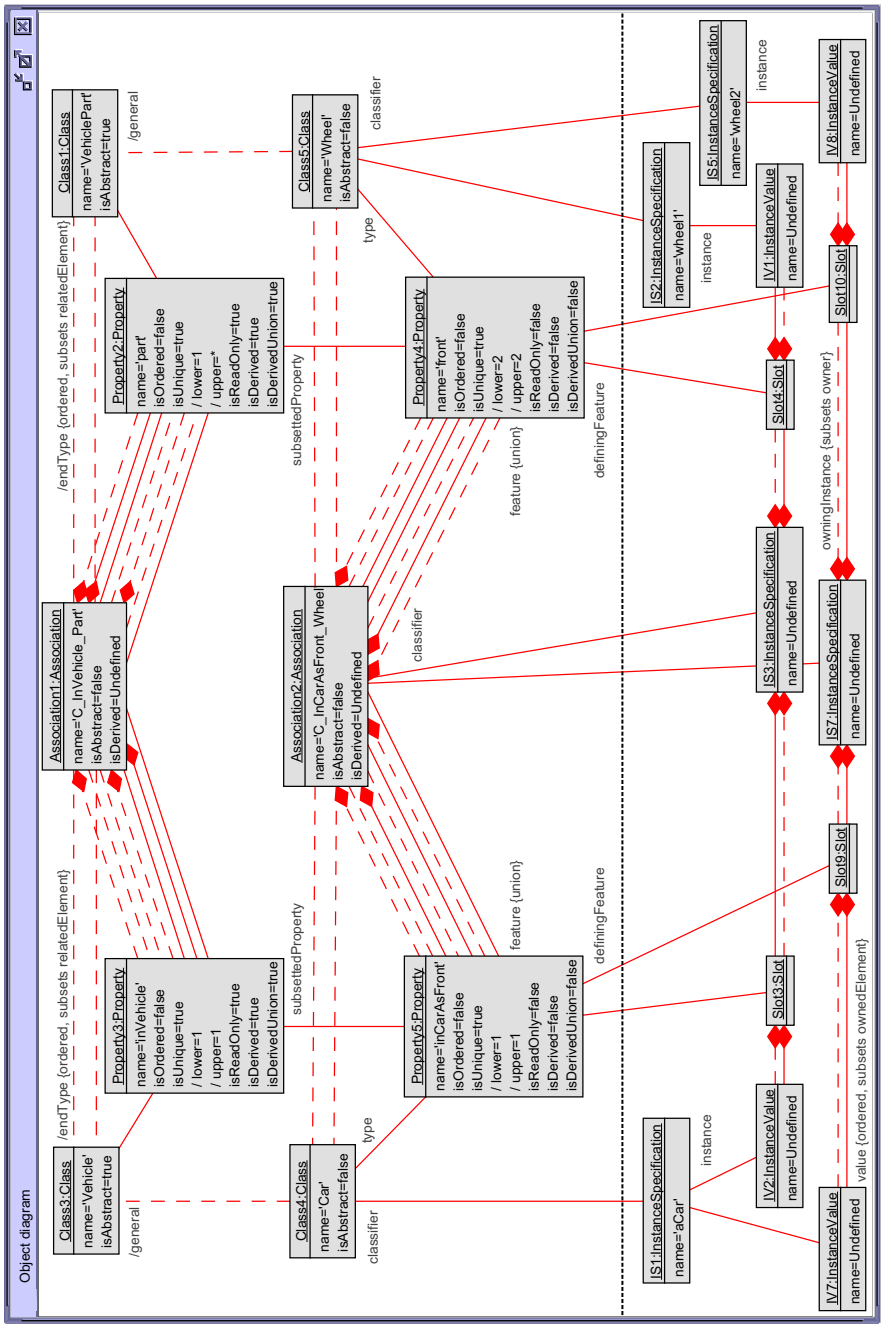

Fig. 6. The diagrams shown in Fig. 4 and 5 as an instantiation of the UML metamodel

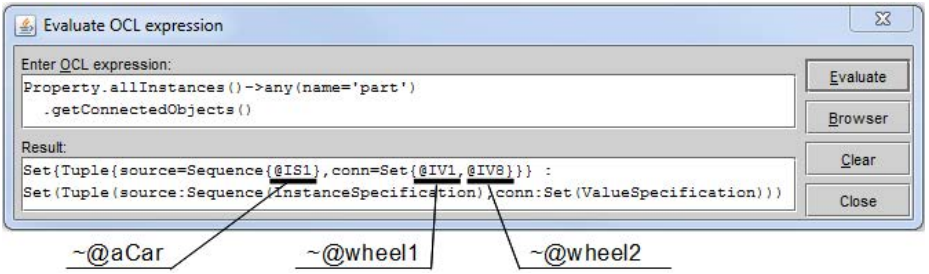

Fig. 7. Querying runtime values by using the operation `getConnectedObjects()`

The result is a set of tuples with two parts:

1. **source**: The sequence of source objects in the same order as the association ends, if the property is owned by an association.
2. **conn**: The objects connected to the source objects at the property.

The result of the evaluation is the calculated union of the property values for all possible source objects. Because only one vehicle (named `aCar`), is present in the given state, the set contains a single tuple. This tuple consists of the sequence containing the instance specification representing the object `aCar` and a set of values which are linked to this instance via subsetting properties of `part`.

Given the previously described operation `getConnectedObjects()`, we can define a constraint which ensures the subsetting semantics:

```
1 context Property inv subsettingIsValid:
2  let subsetLinks = self.getConnectedObjects() in
3  self.subsettedProperty->forAll(supersetProperty |
4    let supersetLinks = supersetProperty.getConnectedObjects() in
5    subsetLinks->forAll(t1 |
6      supersetLinks->one(t2 | t1.source=t2.source and
7        t2.conn.getValue()->asSet()->includesAll(
8          t1.conn.getValue()->asSet()))))
```

The central part of the given invariant can be seen on line 7 where the operation `includesAll` is used, which is the OCL way to validate, if a collection is a superset of another one. Some things need to be explained in a more detail. First, the usage of the operation `getValue():OclAny`, which is an extension to the UML metaclass `ValueSpecification`, is required to be able to get the concrete value of a value specification. The UML metamodel defines several operations on this class for retrieving basic types like `stringValue():String` but excludes a generic definition. Second, the collected values need to be converted to a set using `->asSet()` (see lines 7 and 8) because values can map to the same specifications. It should be mentioned, that if evaluated at runtime, the invariant only validates the union calculation if subsets is used in the context of a derived union. If subsets is used on a property which is not a derived union, the

constraint validates the user defined structure. Including the described invariant and similar invariants for other runtime elements, adds a precise definition of its semantics to the modeling language.

3.2 Derived Properties

Derived properties are widely used during the specification of models and metamodels, because they allow to shorten certain expressions and to assign associated elements an exact meaning by naming them. If a formal expression is given which describes how to calculate the values of the derived properties, the definition of the metamodel is even stronger. If the derived property is marked as read only, a query language can be used to evaluate these derive expressions. Writable derived properties are allowed for example in the UML, but we exclude this type of properties, because the computational overhead of computing the inverse values would be too high. Furthermore, only bijective derive expressions can be used. For example, an attribute `weight` for the class `Car` used in the example could be derived as follows:

```
context Car::weight:Integer derive: self.part.weight->sum()
```

Assigning a value to the attribute weight of a car cannot lead to a single result in the weights of the parts. A common way to overcome this issue is to use a declarative approach like it is done in the UML specification by using invariants for a derive expression [23, p. 128]. This transfers the responsibility to set the correct derived values or the inverse direction to an implementation. Therefore, the UML metamodel excludes the ability to add a derive expression to a property like it is done with default values. Whereas, the OCL specification links to the UML metamodel for the placement of derive expressions [24, p. 182]. We propose to add such a possibility, to allow the specification of the runtime semantics of derived read only properties. For this, we extend the metamodel by defining an additional association between `Property` and `ValueSpecification`. To ensure, that a derived expression is only used on read only properties, the following well-formedness rule needs to be added:

```
context Property inv: self.derivedValue <> null implies self.readOnly
```

The context of such a derive expression used during evaluation is related to the previously explained semantics of `subsets` and `union`. To recapitulate the essentials, for a generic solution it is necessary to consider the combinations of source objects and their connected objects. Only this allows to use derived association ends on associations with more than two association ends and further allows the evaluation of backward navigations, i. e., from a derived end to an opposite end. The major difference to the validation of subsetting is, that only if a derived association end of a binary association or an attribute are the target of a navigation, the source objects are known. If a navigation uses instead the derived end as the source, for all possible combinations of the connected end types the expression needs to be evaluated and checked if the source object of

the navigation is in the result. As an example consider the derived association end `/general` of the reflexive association defined on the class `Classifier` shown in Fig. 1. The UML specification defines the derived end using a constraint on classifier as follows [23, p. 52][3]:

```
general = self.generalization.general->asSet()
```

Used as a derive expression, the result for a navigation from a classifier instance to the association end `general` can be calculated using the source instance as the context object `self`. For the opposite direction of the navigation, i. e., navigating from a classifier instance to its subclasses, the derive expression needs to be evaluated for all instances of `Classifier`:

```
superclass = Classifier.allInstances()->select(general->includes(self))
```

For n-ary associations navigating to the derived association end, the derive expression needs to be evaluated with each combination of the source object and all possible instances at the other ends (excluding the derived end). The resulting set is the union of all evaluation results. If a navigation starts at the derived end of an n-ary association, the calculation is similar to the case of navigating backward in a binary association. Except, that the evaluation is performed for the cross product of all instances which can participate in the association. This means all instances of the end types except the derived end.

4 Tool Based Validation

Because of the endogenous nature of the semantics described in the previous chapter, they were developed in parallel to extensions to a modeling tool. To validate the structural constraints used inside the UML metamodel, these were added to the tool, which allowed us to represent greater parts of the metamodel. Using a tool based validation approach and extending it in a step-wise manner added a reverse link to the specification of the runtime semantics. Without a validation tool, it is rather hopeless to bring a metamodel including well-defined semantics for a modeling language to a consistent state. Using a modeling tool to validate its modeling language, like the bootstrapping approach used for compilers, allows to discover issues beyond syntactical errors in an early state. For example, only after using derived unions in combination with derived association ends we discovered an infinite recursive definitions at the metamodel level in the current UML standard. In this particular case, a derived association end was used inside a union and the derive expression used this union. In the following parts of this section, we explain some beneficial features supporting the definition of (meta-)models which are integrated in our modeling tool USE [11,30]. Additional supporting features are beyond the scope of this paper, but can be

[3] The constraint has slightly been modified to be more expressive. In detail, the body of the operation `Classifier::parent()` was embedded into the constraint. Further, `asSet()` was added to establish type soundness.

found in several publications of our group, e. g., [13,14,12]. Such a left out feature is the possibility to evaluate the specified constraints on a model instance, which was used to validate the invariants presented in this paper.

During the development of a metamodel, already on the syntactical level the usage of automatically generated dynamic views can support the user. While the size of a model increases, the usage of the modeling elements discussed in this paper (subsets, union and derived properties) can get unmanageable without adequate support by a tool. USE provides a comprehensive view which provides information about these elements defined for an association. An example of this view is presented in Fig. 8. It shows the derived union association specified between the metaclasses `Classifier` and `Feature` in the UML metamodel. A user can directly see which associations are related to the selected one and what kind of relations are defined. Implicit information, like for example a missing subsets on the opposite end is highlighted.

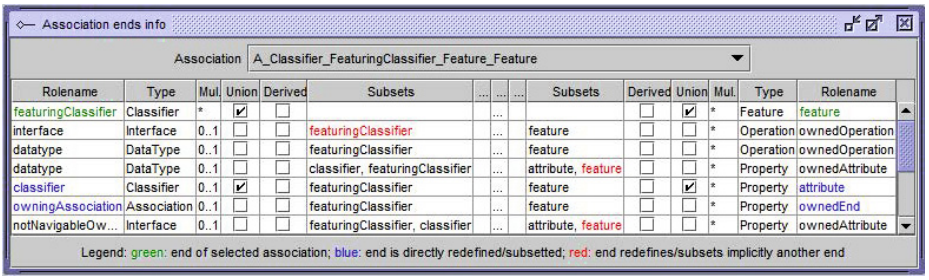

Fig. 8. Information about association relations available in USE

Another valuable functionality, which was touched slightly while explaining Fig. 5 and 6 is the automatic calculation and presentation of virtual links (presented as dashed lines) which result from associations that include a derived expression or derived unions. In Fig. 9 an in-depth view on the defined and derived links between the instances representing the composition `C_InCarAsFront` and its owned end `front` is shown. While the three lower links are specified by the user, the upper four links are automatically presented to the user because they are part of a derived union. Another usage of virtual links is to compress diagrams as it was done in Fig. 6 by excluding the generalization instances, but still showing the generalization link between classes using the derived end `/general`.

Furthermore, using derived associations allows a user to model information in a different way which may be more suitable to express her intention. The USE session presented in Fig. 10 shows an example, which uses a derived ternary association to show the direct relation of associated objects. The example defines a small library model composed of classes for users, copies and books. The fact that a user can borrow copies of books is modeled by two binary associations which together link all three classes. A third association is defined, that is derived

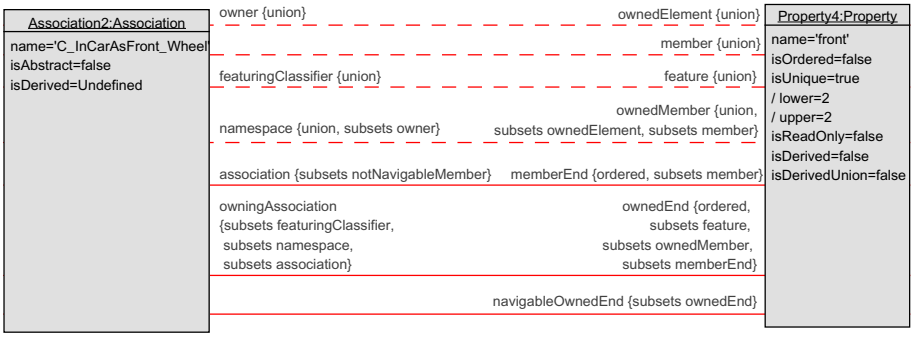

Fig. 9. A detailed view on virtual links present in the UML metamodel instance (Fig. 6)

and combines the aforementioned associations into a single ternary one. The definition of the derived association in the concrete syntax of USE is as follows:

```
association BorrowsCombined between
  User[*] role dUser
  Copy[0..1] role dCopy derived(aUser:User,aBook:Book) =
    aUser.copy->select(c | c.book=aBook)
  Book[*] role dBook
end
```

The shown textual language is an excerpt of the language used to define UML models in USE. It is comparable to HUTN (UML Human-Usable Textual Notation) of the OMG [21]. To be able to show derived links, our language defines the keyword `derive` to mark an an association end as derived. The derive keyword requires an OCL expression which defines the derived links. For n-ary associations, also the naming of the parts of a combination is required to be able to evaluate an arbitrary OCL expression. In contrast to this, a derived expression on a binary association can use a single context variable `self`, because there is no combination of instances at association ends.

For example, to calculate the links for the association `BorrowsCombined` the derive expression at the association end `dCopy` is evaluated for all pairs of `User` and `Book` objects (these pairs are expressed by the signature `(aUser:User, aBook:Book)` of the derive definition shown above. The derive expression returns all copies associated with a given pair of a user and a book. For each `Copy` object in the result set a link connected to the input pair and the copy object is shown in the object diagram. In addition, the example shows how one can use a multiplicity constraint on derived associations. In this example, the multiplicity constraint `0..1` in the association end `dCopy` excludes double borrowings (a user borrows more than one copy of the same book). The multiplicity violation of the example state is reported to the user, as can be seen at the bottom of Fig. 10.

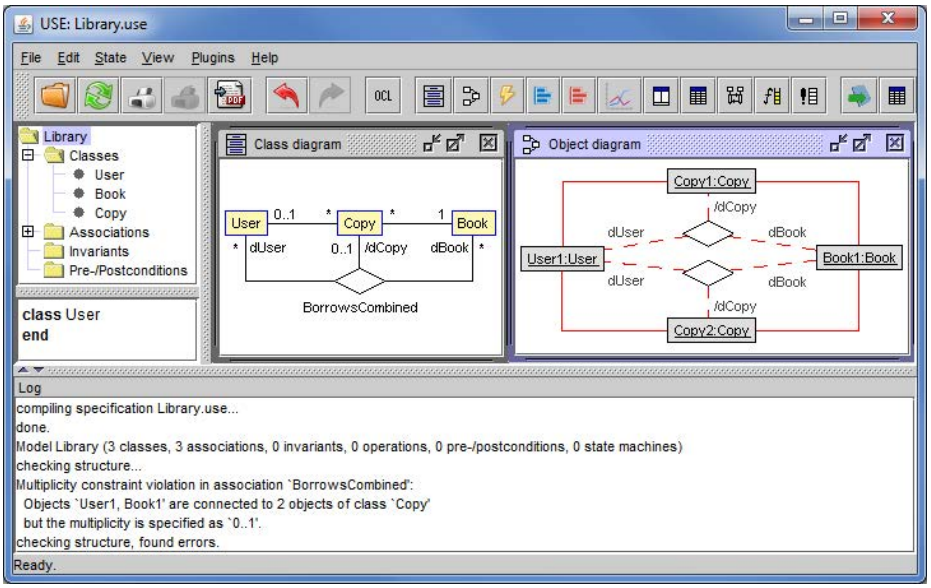

Fig. 10. Screenshot of USE while validating a snapshot with derived ternary association

5 Related Work

Metamodeling semantics has been used in areas not focused in this paper. In [8] it is applied to define the semantics of multiple inheritance using a set-theoretic based metamodel. [16] shows its application to specify the semantics of OCL, whereas [9,15] cover a detailed view on the overall topic of metamodeling semantics. A combined view of different metamodeling levels is used in [10] to specify the semantics of entity relationship diagrams and their transformation into the domain of relational schemata.

As examples for the ongoing discussion about the need of a formal semantics for UML and to what extend it should be defined, we refer to [27] and [5]. The authors of [5] discuss the benefits and drawbacks of a precise UML specification including runtime semantics from several points of view. Furthermore, the problems arising by trying to be a general purpose language for different domains implying semantic variation points is explained. We believe, that both points of view are valid, but the viewpoints change during the development process. At an early stage of design, the used modeling language could allow to violate the precise semantics. While the process continues, these violations should be more and more forbidden until a state is reached where no violation is allowed.

Beside the vast amount of publications defining the semantics of UML, e. g. [18,31,28], work covering the UML language elements presented in this paper has been done. [4] gives a descriptive insight of using union and subsets and shows its relation to composite structures.

Exogenous definitions of the semantics for subset and union properties have, for example, been provided in [1] using a set-theoretic formalization, [3,2] using graph transformations, and [19] using a so-called property oriented abstract syntax to define the semantics of what the authors call inter-association constraints (these include subsets and union). These examples of exogenous definitions of semantics all require to have expertise in the respective external semantic technology space. [20] introduces a UML profile covering redefinition and other elements. While the work is similar to ours in the sense that it stays in the same technological space, the runtime semantics is enforced generating model specific OCL constraints, like the ones shown at the beginning of Sect. 3. A semantics for subsetting using the same transformation approach is given in [7]. Another transformation approach to describe the runtime semantics of UML constraints using OCL is shown in [6]. Here, the runtime semantics implied by UML compositions are translated to OCL constrains, i. e. the semantics must be defined by a transformation into a specific application model. Whereas our semantics works in a universal way, where constraints are formulated on the metamodel level without the need for transformation.

In this paper we presented a way to validate (meta-)model instances by creating snapshots, i. e., instantiations, of these models and by examining their behavior, for example, by checking the multiplicity constraints on an instance or by examining the current states of the defined invariants. Other approaches use automatic techniques to reason about models specified in UML/OCL. An approach like [17] could, for example, be used to find valid configurations of writable derived properties as discussed earlier in this paper. In addition, it can be used like the ones in [29] and [25] to answer questions about the satisfiability and other properties of a model.

6 Conclusion and Future Work

We presented a proposal to specify the runtime semantics of a modeling language using a metamodel describing syntax and semantics in the same language. Using the same technology space reduces the overall complexity of the language description, because knowledge of other languages is not required. Furthermore, the process of specifying the language is improved, if this self describing technique is used in combination with tool-supported validation. As we have shown in Sect. 4, bringing models into being by creating snapshots can give insights into the model which are rather vague if only the static specification is used.

As future work, the application of our approach to other areas of modeling languages, for example property redefinition and association generalization, seem to be promising directions to extend our work. The covered elements of the UML metamodel for validation and the options on the user interface in our tool USE can be strengthened as well. Larger case studies with other modeling language, for example domain-specific languages, will give further feedback on the usability of the approach.

References

1. Alanen, M., Porres, I.: A metamodeling language supporting subset and union properties. Software and Systems Modeling 7(1), 103–124 (2008)
2. Amelunxen, C.: Metamodel-based Design Rule Checking and Enforcement. Ph.D. thesis, Technische Universität Darmstadt (2009), dissertation
3. Amelunxen, C., Schürr, A.: Formalizing Model Transformation Rules for UML/MOF 2. IET Software Journal 2(3), 204–222 (2008); Special Issue: Language Engineering
4. Bock, C.: UML 2 Composition Model. Journal of Object Technology 3(10), 47–73 (2004), http://www.jot.fm/issues/issue_2004_11/column5
5. Broy, M., Cengarle, M.V.: UML formal semantics: lessons learned. Software and System Modeling 10(4), 441–446 (2011)
6. Chavez, H.M., Shen, W.: Formalization of UML Composition in OCL. In: Miao, H., Lee, R.Y., Zeng, H., Baik, J. (eds.) ACIS-ICIS, pp. 675–680. IEEE (2012)
7. Costal, D., Gómez, C., Guizzardi, G.: Formal Semantics and Ontological Analysis for Understanding Subsetting, Specialization and Redefinition of Associations in UML. In: Jeusfeld, M., Delcambre, L., Ling, T.-W. (eds.) ER 2011. LNCS, vol. 6998, pp. 189–203. Springer, Heidelberg (2011)
8. Ducournau, R., Privat, J.: Metamodeling semantics of multiple inheritance. Science of Computer Programming 76(7), 555–586 (2011)
9. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In: Evans, A., Caskurlu, B., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 323–337. Springer, Heidelberg (2000)
10. Gogolla, M.: Exploring ER and RE Syntax and Semantics with Metamodel Object Diagrams. In: Nürnberg, P.J. (ed.) Proc. Metainformatics Symposium (MIS 2005). ACM Int. Conf. Proceeding Series, vol. 214, 12 pages. ACM Press, New York (2005)
11. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69, 27–34 (2007)
12. Gogolla, M., Hamann, L., Xu, J., Zhang, J.: Exploring (Meta-)Model Snapshots by Combining Visual and Textual Techniques. In: Gadducci, F., Mariani, L. (eds.) Proc. Workshop Graph Transformation and Visual Modeling Techniques (GTVMT 2011). ECEASST, Electronic Communications (2011), journal.ub.tu-berlin.de/eceasst/issue/view/53
13. Hamann, L., Hofrichter, O., Gogolla, M.: OCL-Based Runtime Monitoring of Applications with Protocol State Machines. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 384–399. Springer, Heidelberg (2012)
14. Hamann, L., Hofrichter, O., Gogolla, M.: On Integrating Structure and Behavior Modeling with OCL. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MoDELS 2012. LNCS, vol. 7590, pp. 235–251. Springer, Heidelberg (2012)
15. Hausmann, J.H.: Dynamic META modeling: a semantics description technique for visual modeling languages. Ph.D. thesis, University of Paderborn (2005)
16. Kleppe, A.: Object constraint language: Metamodeling semantics. In: Lano, K. (ed.) UML 2 Semantics and Applications, pp. 163–178. John Wiley & Sons, Inc. (2009)

17. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 290–306. Springer, Heidelberg (2011)
18. Lano, K.: UML 2 Semantics and Applications. John Wiley & Sons, Inc. (2009)
19. Maraee, A., Balaban, M.: Inter-association Constraints in UML2: Comparative Analysis, Usage Recommendations, and Modeling Guidelines. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MoDELS 2012. LNCS, vol. 7590, pp. 302–318. Springer, Heidelberg (2012)
20. Nieto, P., Costal, D., Gómez, C.: Enhancing the semantics of UML association redefinition. Data Knowl. Eng. 70(2), 182–207 (2011)
21. OMG (ed.): UML Human-Usable Textual Notation (HUTN). Object Management Group (OMG) (August 2004), http://www.omg.org/spec/HUTN/
22. OMG (ed.): Meta Object Facility (MOF) Core Specification 2.4.1. Object Management Group (OMG) (August 2011), http://www.omg.org/spec/MOF/2.4.1
23. OMG (ed.): UML Superstructure 2.4.1. Object Management Group (OMG) (August 2011), http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF
24. OMG (ed.): Object Constraint Language 2.3.1. Object Management Group (OMG) (January 2012), http://www.omg.org/spec/OCL/2.3.1/
25. Queralt, A., Teniente, E.: Verification and Validation of UML Conceptual Schemas with OCL Constraints. ACM Trans. Softw. Eng. Methodol. 21(2), 13 (2012)
26. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language - Reference Manual, 2nd edn. Addison-Wesley (2004)
27. Rumpe, B., France, R.B.: Variability in UML language and semantics. Software and System Modeling 10(4), 439–440 (2011)
28. Shan, L., Zhu, H.: Unifying the Semantics of Models and Meta-Models in the Multi-Layered UML Meta-Modelling Hierarchy. Int. J. Software and Informatics 6(2), 163–200 (2012)
29. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 152–170. Springer, Heidelberg (2011)
30. A UML-based Specification Environment. Internet, http://sourceforge.net/projects/useocl/
31. Varró, D., Pataricza, A.: Metamodeling Mathematics: A Precise and Visual Framework for Describing Semantics Domains of UML Models. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 18–33. Springer, Heidelberg (2002)
32. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Object Technology Series. Addison-Wesley, Reading (2003)