Correctly Defined Concrete Syntax for Visual Modeling Languages

Thomas Baar

École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
thomas.baar@epfl.ch

Abstract. The syntax of modeling languages is usually defined in two steps. The abstract syntax identifies modeling concepts whereas the concrete syntax clarifies how these modeling concepts are rendered by visual and/or textual elements. While the abstract syntax is often defined in form of a metamodel there is no such standard format yet for concrete syntax definitions; at least as long as the concrete syntax is not purely text-based and classical grammar-based approaches are not applicable. In a previous paper, we proposed to extend the metamodeling approach also to concrete syntax definitions. In this paper, we present an analysis technique for our concrete syntax definitions that detects inconsistencies between the abstract and the concrete syntax of a modeling language. We have implemented our approach on top of the automatic decision procedure Simplify.

1 Introduction

The trend to model-driven development is facing the question how modeling languages can be defined precisely in a standardized format. Metamodeling is today the prevailing technique in order to define the abstract syntax of modeling languages in a precise, non-ambiguous way: metaclasses represent all modeling concepts, metaattributes their variations, metaassociations their relationships. Well-formedness rules written as OCL invariants insure that certain conditions are satisfied in all syntactically correct sentences of the modeling language. The abstract syntax definition is the most basic block when defining a modeling language but, at the same time, it is the only block for which a commonly agreed format exists. All other blocks of a modeling language definition, e.g. the definition of concrete syntax and the definition of semantics, are given in many cases only informally. A prominent example for an informal language definition is UML, see [1]. The most important disadvantage of informal definitions is the lack of tool support for checking the consistency of the definition.

This paper is about formal *concrete syntax* definitions for modeling languages having a visual, i.e. not purely textual, notation. In the first part (Sect. 2 and Sect. 3), we briefly describe a metamodeling approach to define not only the abstract syntax but also the concrete syntax of a modeling language formally (this approach has been already presented with more details in [2]). As an illustration, we use UML class diagrams, mainly, because class diagrams have a well-known visual concrete syntax. In the paper's main part (Sect. 4), we show how concrete syntax definitions can be analyzed rigorously and automatically checked. Intuitively, the concrete syntax is ill-defined if two different models (i.e. instances of the abstract syntax metamodel) can be rendered by the same diagram. As a tiny example, one can take UML class diagrams whose classes can be abstract and non-abstract according to the abstract syntax. Suppose, the concrete syntax would only stipulate to render each class by a rectangle and to label the rectangle with the name of the class. Then, one could not infer from a given diagram whether a rectangle represents an abstract or a non-abstract class and this ambiguity is an error of the concrete syntax definition.

Such errors can be automatically detected by using deductive tools. More precisely, we generate out of a formal concrete syntax definition a proof obligation that is valid if and only if the syntax definition does not contain any error. Then, this proof obligation is passed to the deductive tool Simplify [3], which was able to automatically discharge all of them for the examples given in this paper.

2 Visual Languages

Modeling languages having a visual concrete syntax use for the representation of models graphical elements such as rectangles, circles, lines, stickmen, etc. Graphical elements are also called *visual objects* since they can easily be described as objects whose state is given by the value for certain attributes such as *shape*, *lineColor*, *backgroundColor*, *attachRegion*, etc. A set of visual objects is a syntactically correct *sentence of a visual language* when all well-formedness rules of the visual language are met. A typical example for a well-formedness rule is that a visual object of shape *Line* always connects two other visual objects, more technically, that the start- and endpoint of the line coincide with the attach regions of the connected visual objects. We call a syntactically correct sentence of a visual language also *diagram*.

The definition of a visual language is done in two steps: (1) identification of all attributes for visual objects, and (2) formulation of well-formedness rules. For non-trivial visual languages, it is worthwhile to distinguish classes of visual objects because not all possible attributes are relevant for each object, e.g. a visual object of shape *Line* does not need a value for an attribute *backgroundColor*. Once the classes of visual objects together with their attributes are identified, many well-formedness rules of the visual language can easily be expressed by associations between classes. For example, the above given restriction for a line to connect two other visual objects is best expressed by two associations from class `Line` to a class, let's say, `ConnectableObject` (which represents the connected visual objects) with multiplicity 1 at the latter class.

Having said this, it is obvious that a metamodel is a very appropriate format to define a visual language formally. A diagram is then just an instance of the metamodel of the visual language.

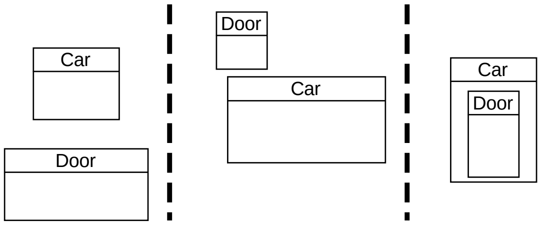


Fig. 1. Three diagrams – when read as representations of class diagrams, the first two diagrams should not be distinguishable

As an example, we discuss the three diagrams given in Fig. 1. What we see – at a first glance – are two labeled rectangles, which have in all three diagrams different dimensions and different positions. An initial version of the metamodel could consist of one class `Rectangle` with attributes for (1) the label, (2) the x and y coordinates of the position, and (3) the dimension (width, height). According to this metamodel, all three diagrams are different. This initial metamodel is very suitable if the layout information of the diagrams have to be captured; for instance, when diagramming tools have to exchange diagrams. Actually, the initial metamodel can be seen as a drastically simplified version of the upcoming OMG standard for Diagram Interchange [4]. However, the initial metamodel is less useful as a basis for a concrete syntax definition for class diagrams. When read as class diagrams, the left and middle diagram should coincide, despite the fact, that the dimensions and positions of the two rectangles are different. While the right diagram also shows the same classes as the first two, just the position and the dimension were changed again, it is nevertheless semantically different from the others.[1]

What this tiny example already shows is the fact, that layout information in form of coordinates and dimensions are not necessary for the definition of a concrete syntax. It is better to choose such attributes for visual objects that reflect differences of rendered models. For example, we cannot fully ignore layout information because this would make all three diagrams in Fig. 1 non-distinguishable.

Figure 2 shows a more suitable metamodel for the visual language used in Fig. 1. The class `Rectangle` has again one attribute for label but none for position and dimension. In order to distinguish the last diagram from the two others, a self-association on `Rectangle` has been introduced that encodes graphical nesting of rectangles. In the lower part of Fig. 2, the three diagrams from Fig. 1 are

[1] When read as a UML class diagram, the graphical containment of class `Door` in class `Car` means that `Car` is composed of `Door`.

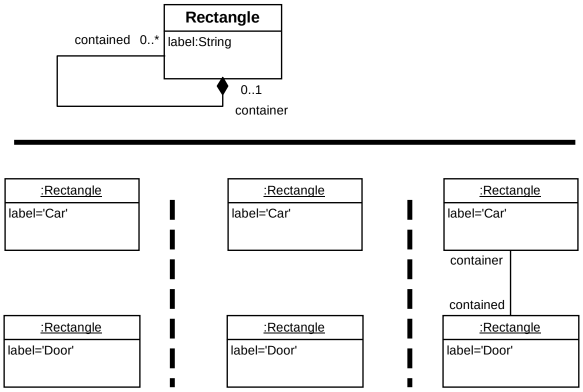


Fig. 2. Visual language definition and representation of diagrams given in Fig. 1

given as instances of the visual language metamodel and the first two diagrams coincide indeed.

The definition and efficient processing of visual languages is a current research area, which we cannot develop further here due to space limit. A warmly recommended introduction is [5] where a classification of visual languages is presented and formats for elegant language definitions are derived. A core technique, which has been also applied in the above example, is to substitute absolute layout information (such as position, dimension) by relative ones, called *spatial relationships*. When defining a metamodel for a visual language, one has to identify – in a first step – all relevant spatial relationships. For example, the rendering of UML models requires graphical nesting as one spatial relationship but there are other spatial relationships needed as well.

3 Concrete Syntax Definition

In the previous section, we have outlined how a visual language can be formalized in form of a metamodel; we will now answer the question how sentences of a modeling language, which are given as instances of the abstract syntax metamodel, can be rendered in this visual language. The missing part is, informally speaking, the bridge from the abstract syntax metamodel to the metamodel of the visual language. In the following, we describe briefly our approach to define the concrete syntax and illustrate it on a fragment of UML class diagrams. The approach of *defining* the concrete syntax has been already described in one of our previous papers [2] and was recently implemented based on SVG technology [6]. The core idea for bridging both metamodels is to introduce new classes in between. This technique is well-known from Triple-Graph-Grammars [7] and is also applied in the OMG standard for Diagram Interchange [4].

Figure 3 gives an overview on the structure of concrete syntax definitions. In the left part, a metamodel for the abstract syntax is shown: each instance

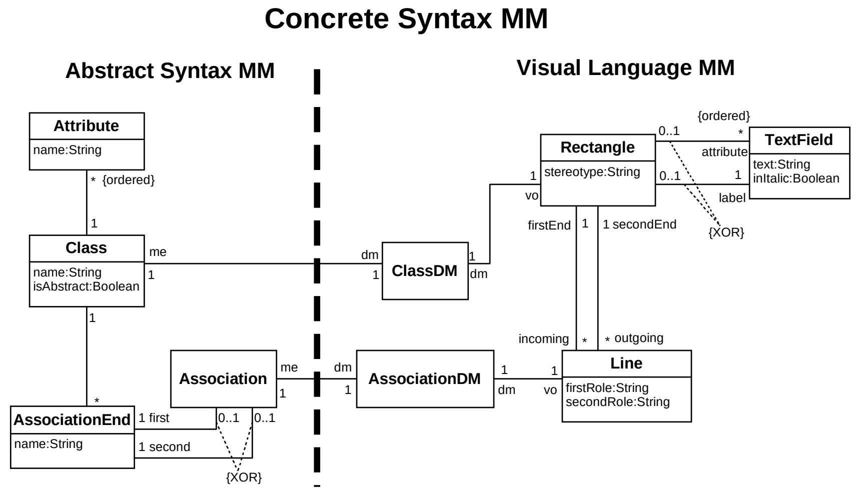


Fig. 3. Bridging the metamodels describing abstract syntax and visual language

of `Class` is connected to a sequence of `Attribute` instances and instances of `Association` have two `AssociationEnd`s which refer to exactly one `Class`. The metamodel of the visual language is shown in the right part and describes graphical elements like rectangles, lines and text fields.

The two classes `ClassDM` and `AssociationDM` are so-called *display manager classes* (the name of these classes has, by convention, always the suffix `DM`) and realize the bridge from the abstract syntax to the visual language. Strictly speaking, display manager classes belong neither to the metamodel of the abstract syntax nor to that of the visual language since they are added later on, when the concrete syntax is defined. For our argumentation, however, it has advantages if they are seen as part of the metamodel of the visual language. A display manager class is always connected via an association with multiplicity 1-1 to a class from the abstract syntax metamodel. The display manager class manages the rendering of the referenced class. By convention, we always use `me` (for **m**odel **e**lement) and `dm` (for **d**isplay **m**anager) as role names on this association. Display manager classes have also an association to a class in the visual language metamodel. Usually, this association has multiplicity 1-1 as well and role names `dm` and `vo` (for **v**isual **o**bject).

The bridge from the abstract syntax to the visual language is realized by invariants that are attached to the display manager classes. These invariants formalize synchronization conditions on the states of modeling elements and the corresponding visual objects (which realize the rendering of the modeling elements). For our example, the invariants are:

```
context AssociationDM inv:
        self.me.first.name=self.vo.firstRole
  and   self.me.second.name=self.vo.secondRole
  and self.vo.firstEnd=self.me.first.class.dm.vo
  and self.vo.secondEnd=self.me.second.class.dm.vo

context ClassDM inv:
      self.me.name=self.vo.label.text
  and (self.me.isAbstract implies
        (self.vo.stereotype='abstract' or
         self.vo.label.inItalic))
  and (not(self.me.isAbstract) implies
        (self.vo.stereotype='' and
         not(self.vo.label.inItalic)))
  and self.me.attribute->size()=self.vo.attribute->size()
  and Set{1..self.me.attribute->size()}->forAll(i|
        self.me.attribute->at(i).name=
          self.vo.attribute->at(i).text)
```

Based on Fig. 3 and the invariant for `AssociationDM` one can conclude, that each instance of `Association` is rendered by a `Line`, whose annotations `firstRole` and `secondRole` correspond to the names of the two association ends. Furthermore, the line connects the two rectangles which render the classes the two association ends are referring to. The invariant for `ClassDM` is slightly more complicated since it allows for *presentation options* when rendering a class. An abstract class can be marked by a stereotype 'abstract' attached to the corresponding rectangle or the label of the rectangle is displayed in an italic font. The attributes of a class are presented in the same order as textfields in the rectangle. For the rendering of the attributes it does not matter whether they are set in italic or not, they just represent attributes that are given by their names.

To summarize, our approach to define concrete syntax

- describes in a declarative way all possible representations of models (instances of the abstract syntax metamodel) by diagrams (instances of the visual language metamodel). Note that our technique allows to define presentation options, i.e. one model can be rendered by different, i.e. non-isomorphic, diagrams. But – since the concrete syntax definition is symmetric – the opposite case that one diagram renders different, i.e. non-isomorphic, models is possible as well. Such a concrete syntax definition would be incorrect (a diagram should always represent only one model) and in Sect. 4 we will discuss an approach to detect such incorrect concrete syntax definitions.
- does not require to define a display manager class for all classes of the abstract syntax metamodel. In our example, display manager classes are defined only for `Class` and `Association` whereas for `Attribute`, `AssociationEnd` this was not necessary, since the rendering of these classes are captured by `ClassDM`, `AssociationDM` as well. We will give in Sect. 4.2 a detailed analysis, under which circumstances a class from the abstract syntax metamodel does not need its own display manager class.

4 Analysis of Concrete Syntax Definitions

As already mentioned in the introduction and at the end of the last section, concrete syntax definitions can be incorrect. Correctness basically[2] means in our context that each diagram must correspond to only one model. As a tiny example for an incorrect concrete syntax definition we refer to the upper part of Fig. 4. Suppose, that the display manager class `ClassDM` has attached the following invariant:

```
context ClassDM inv:
    self.me.name=self.vo.text
```

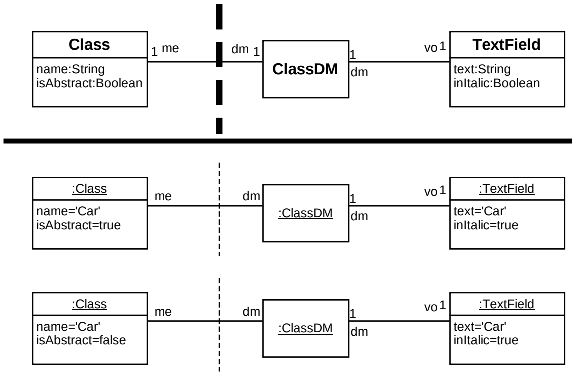


Fig. 4. Incorrect syntax definition and counterexample

The lower part of Fig. 4 shows two instantiations that conform to all multiplicity constraints and to the invariant for `ClassDM`. These instantiations witness an error in the concrete syntax definition since they show how two isomorphic diagrams refer to two non-isomorphic models. If the user of an editor would draw one of the diagrams, he could not be sure which of the two possible models this diagram actually represents. The instantiations are possible because the invariant attached to `ClassDM` only stipulates how attribute `name` of the model element is related to attribute `text` of its visual representation but ignores the value of attribute `isAbstract`.

The correctness criterion for concrete syntax definitions is given with mathematical rigor by the following definition:

Definition 1 (Correctness of Concrete Syntax Definitions). *Let CSMM be a concrete syntax definition given in form of a metamodel (cmp. Fig. 3). Since*

[2] There is another criterium on the completeness of the concrete syntax definition saying that for each model there is at least one diagram. However, this is not discussed in this paper.

CSMM is divided into two parts describing abstract syntax and visual language, this division can also be applied to instances of CSMM. Let cs1, cs2 be two instances of CSMM. We denote the part of cs1/cs2 belonging to the abstract syntax part of CSMM as as1/as2 and the part belonging to the visual language part as vl1/vl2.

We call the concrete syntax definition CS correct *(or* well-defined*) if and only if the following holds:*

Whenever vl1 is isomorphic to vl2 then as1 must also be isomorphic to as2.

In the sequel, we show how this correctness criterion can be encoded into first-order logic so that the decision procedure SIMPLIFY can prove or disprove the generated proof obligation. SIMPLIFY was originally developed to decide the validity of a given formula in the theory of *Pressburger Arithmetik* [8], a set of axioms defining the arithmetic operators for natural numbers except multiplication. SIMPLIFY is also applicable to prove validity in any other first-order theory, but then, due to the undecidability of first-order logic, SIMPLIFY is not able to prove all valid theorems. For the proof obligations that has been generated as the encoding of our correctness criterion, however, SIMPLIFY was impressively powerful and could prove or disprove every proof obligation for all examples we discuss in this paper. A very useful feature of SIMPLIFY is, that it gives back a counterexample if the proof goal has been disproven. This happens when the concrete syntax definition is erroneous and the generated proof obligations are not valid.

4.1 Encoding of Proof Obligations into First-Order Logic

In this subsection, we justify our encoding of the proof obligations for the most simple kind of syntax definitions, in which the metamodel of the abstract syntax consist of one class only (the definition given in Fig. 4 will serve as an illustrating example). The goal of our argumentation is to justify, that an encoding of the above given correctness criterion into first-order logic is possible. Note that the criterion given in Def. 1 refers to the isomorphism of graphs, a property that can usually not be expressed using first-order logic. Fortunately, in our case, the graphs have a unique structure, which simplifies the encoding of graph isomorphism so that first-order logic has sufficient expressive power. In the next subsection we will present a heuristic on how a concrete syntax definition with more than one class in the abstract syntax part can be reduced to the case we discuss now, where the abstract syntax part has merely one class.

For the rest of this subsection, we assume a concrete syntax definition as illustrated by Fig. 4: The abstract syntax part has only one class (`Class`) that is connected by an 1-1 association with a display manager class (`ClassDM`) that in turn is connected to other classes in the visual language part, in our example we have a 1-1 association to class `TextField`.

The correctness criterion given in Def. 1 requires to check that two isomorphic instances $vl1$, $vl2$ of the visual language part are always connected to isomorphic instances $as1$, $as2$ of the abstract syntax part. The situation is sketched in Fig. 5.

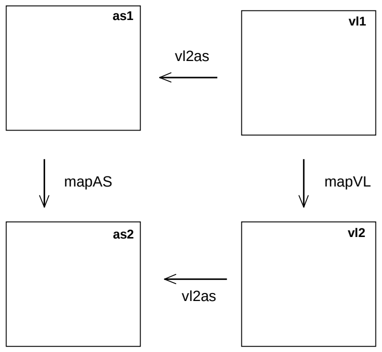


Fig. 5. Bijections that justify correctness criterion

We can assume that $vl1$ is isomorphic to $vl2$, that is, it exists a bijection $mapVL$ that maps in particular each display manager object, i.e. each instance of display manager class `ClassDM`, in $vl1$ to an isomorphic instance in $vl2$. For the display manager objects in $vl1$ and $vl2$ we further know that there is an isomorphism to the objects in $as1$ and $as2$ (because the display manager class `ClassDM` and the abstract syntax class `Class` are connected by an association with multiplicity 1-1). We call this mapping $vl2as$. Based on $mapVL$ and vl2as we can now define a function $mapAS$ as follows (variable cdm represents all instances of `ClassDM` in $vl1$):

$$mapAS(vl2as(cdm)) = vl2as(mapVL(cdm))$$

Please note that $mapAS$ is defined as a total function from $as1$ into $as2$, because each object in $as1$ has a corresponding display manager object in $vl1$.

If we could show now that $mapAS$ maps the objects from $as1$ to *isomorphic* objects in $as2$ then this would prove that $as1$ and $as2$ themselves (both are sets of objects) are isomorphic what in turn would complete the proof on the correctness of the concrete syntax definition.

It remains to show for each isomorphism $mapVL$ between $vl1$ and $vl2$ that the derived function $mapAS$ is an isomorphism too (cdm is again a variable of type `ClassDM`):

$$isIsomorphicClassDM(cdm, mapVL(cdm)) \rightarrow$$
$$isIsomorphClass(vl2as(cdm), mapAS(vl2as(cdm)))$$

According to the above given definition of $mapAS$, this can be simplified to:

$$isIsomorphicClassDM(cdm, mapVL(cdm)) \rightarrow$$
$$isIsomorphClass(vl2as(cdm), vl2as(mapVL(cdm)))$$

Fortunately, this proof obligation does not require anymore to formulate the isomorphism of the whole graph but just to specify the isomorphism of two objects, a property for which first-order logic is expressive enough. For instance, two objects of `ClassDM` are isomorphic if their attributes have the same values and the connected `TextField` objects are isomorphic. Since `ClassDM` and `TextField` are connected by a 1-1 association, the latter means that two isomorphic instances of `ClassDM` are always linked to two instances of `TextField` whose attributes have also the same value. Formulated in first-order logic, the criteria for isomorphic `ClassDM` instances looks like:

$$\begin{aligned}
&isIsomorphicClassDM(cdm1, cdm2) \leftrightarrow \\
&\quad (text(vo(cdm1)) = text(vo(cdm2))\wedge \\
&\quad (inItalic(vo(cdm1)) \leftrightarrow inItalic(vo(cdm2))))
\end{aligned}$$

```
\sorts {
   class;
   classdm;
   textfield;
   string;
}
\functions{
  // associations
  class me(classdm);
  classdm dm(class);
  textfield vo(classdm);
  // attributes
  string name(class);
  string text(textfield);
}
\predicates{
  // attributes
  isAbstract(class);
  inItalic(textfield);
  // predicates to encode isomorphism
  isIsomorphicClass(class,class);
  isIsomorphicClassDM(classdm,classdm);
}
\problem {
// invariant on ClassDM (core of syntax definition)
(\forall classdm cdm; name(me(cdm)) = text(vo(cdm))) &
// isomorphism of instances of Class
(\forall class c1;\forall class c2; (isIsomorphicClass(c1,c2)
         <-> name(c1) = name(c2) &
             (isAbstract(c1) <-> isAbstract(c2)))) &
// isomorphism of instances of ClassDM
(\forall classdm cdm1; \forall classdm cdm2;
         (isIsomorphicClassDM(cdm1,cdm2)
         <-> text(vo(cdm1)) = text(vo(cdm2)) &
             (inItalic(vo(cdm1)) <-> inItalic(vo(cdm2)))))
->
// conclusio
\forall classdm cdm1;\forall classdm cdm2;
  (isIsomorphicClassDM(cdm1,cdm2) -> isIsomorphicClass(me(cdm1), me(cdm2)))
}
```

Fig. 6. Encoding of correctness criterion for Simplify in KeY format

Figure 6 shows the full encoding of the correctness criterion for the example given in Fig. 4. We do not show here the final input file for Simplify, because such input files have to be written in a low level notation, which is hard to read for humans. What is shown here is an input file for the KeY system [9], which can be used as a front-end for Simplify since the KeY system is able to generate automatically equivalent input files for Simplify.

The KeY syntax requires to declare at the beginning of the file all types, functions and predicates. There are standard techniques how an UML class diagram is represented by such declarations, mainly, the classes are represented by types, associations by functions and attributes by functions or predicates (see [9] for details). The clause 'problem' contains the proof obligation and has always the form of an implication *premise* `->` *conclusio*. In KeY syntax, the logical connectors 'not', 'and', 'or', 'if-then', 'if-and-only-if' are denoted by '`!`', '`&`', '`|`', '`->`',

’<->’, respectively, and the two quantifiers are written as ’forall’, ’exists’. The premise of the proof obligation contains the encoding of the invariant of the display manager class ClassDM and the isomorphism criteria for instances of ClassDM and Class. The conclusio has exactly the form as analyzed above.

When invoked for this input file, Simplify cannot find a proof because the syntax definition, for which the input file encodes the correctness criterion, is not correct. Nevertheless, Simplify gives very useful feedback in form of a counterexample. The found counterexample is exactly the same counterexample as we have already presented in the lower part of Fig. 4. Such counterexamples are extremely useful for the developer of the concrete syntax to find and to resolve errors in the concrete syntax definition.

There are (theoretically) two possibilities to fix errors in a syntax definition. As the first possibility, one could refine the visual language or change the constraints attached to the display manager classes. In our example, it would be sufficient to rewrite the invariant of ClassDM to

```
context ClassDM inv:
  self.me.name=self.vo.text and
  self.me.isAbstract = self.vo.inItalic
```

A second possibility is to add to the abstract syntax metamodel a new well-formedness rule but this of course changes the original abstract syntax definition. The idea behind is to avoid the occurrence of all those models that could be cause ambiguous interpretations of the diagrams. An example for such a well-formedness rule is

```
context Class inv:
  self.isAbstract = true
```

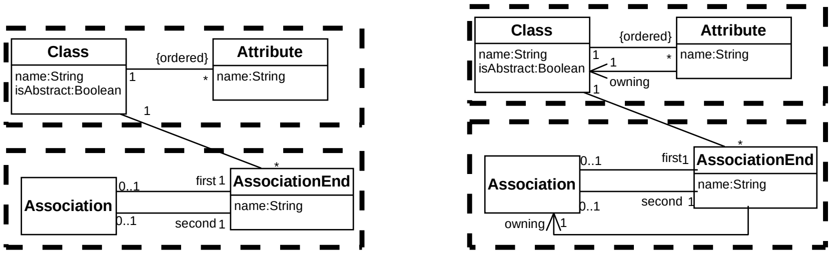
In both cases, Simplify is now able to prove the proof obligation fully automatically what certifies the correctness of the concrete syntax definition.

4.2 Analysis of Complex Syntax Definitions

The encoding presented in the last section covers only the case where the abstract syntax metamodel consists of merely one class. Fortunately, the same encoding also works for abstract syntax metamodels having more than one class, as long as all classes are not connected by any association and each class has its own display manager class in the visual language metamodel.

We discuss now, under which circumstances our encoding is also applicable to more complex abstract syntax metamodels, where classes are connected by associations and not every class has its own display manager class. The basic idea, however, remains the same as in the above case where the abstract syntax metamodel consists only of isolated classes: We strive to find a cluster of classes, i.e. a set of class groups, that induce a partition of the metamodel. Then, we apply our encoding for each of these class groups separately. We illustrate our analysis with the syntax definition for simplified class diagrams as shown in Fig. 3.

In order to find a useful cluster of classes we mark all classes that have a direct connection to a display manager class. The display manager class does not manage only the rendering of the directly connected class, but sometimes also the rendering of the neighboring classes, e.g. `ClassDM` manages the rendering for the instances of both `Class` and `Attribute`. The relevant neighboring classes together with the class directly connected to a display manager form one *class group* in the cluster. At the end of this cluster analysis, we get a situation as shown in the left part of Fig. 7. The cluster consists of two class groups (`Class`, `Attribute`) and (`Association`, `AssociationEnd`) and each class group corresponds to exactly one display manager class (`ClassDM`, `AssociationDM`).



(a) original metamodel

(b) after ownership annotation has been added

Fig. 7. Cluster analysis for abstract syntax classes

The cluster will be the basis for the formal proof that the concrete syntax definition is correct. The formal proof, however, can only be successful if the cluster satisfies two properties, *completeness* and *unique ownership*. Basically, these two properties ensure that all instances of abstract syntax classes can be uniquely mapped to display manager objects which are responsible for the rendering of these instances.

Completeness The found cluster must cover all non-abstract classes, i.e. each non-abstract class must be a member of (at least) one class group. If a class is not a member of any group then the instances of this class do not have any connection to any display manager object.

Unique ownership The completeness criterion is a necessary but not a sufficient condition for the cluster. Sometimes, even instances of classes covered by the cluster miss a corresponding display manager object. Suppose, in our running example the association between `Class` and `Attribute` had on the side of `Attribute` not the multiplicity 1 but '0..1'. That would allow, that some instances of `Attribute` had no connection to any `Class` instance and thus also the connection to a display manager object would be missing. In this case, the current concrete syntax definition would be incorrect, just for structural reasons.[3]

In order to prevent the case, in which an object of any abstract syntax class has no connection to a display manager object, we change the abstract syntax metamodel as follows. In each class group there is exactly one class, called *anchor class*, that has a direct connection to a display manager class (in our example, anchor classes are `Class` and `Association`). We add from each non-anchor class an association with role name 'owning' and multiplicity 1 to the anchor class of the same class group. Furthermore, this association must be derived and the referenced object has to be determined by a constraint.

In our running example, we have added associations from `Attribute` to `Class` and from `AssociationEnd` to `Association` (see right part of Fig. 7). The constraints are

```
context Attribute inv: self.owning=self.class

context AssociationEnd inv: self.owning=
  Association.allInstances()->select(as|
    as.first=self or as.second=self)->any()
```

After the properties *Completeness* and *UniqueOwnership* have been verified for the cluster, we know that each instantiation of the abstract syntax metamodel can be partitioned with respect to the class groups identified by the cluster. What remains to do, is to define predicates for the isomorphism between instances of the same class group.

```
// isomorphism of instances of Class
(\forall class c1;\forall class c2; (isIsomorphicClass(c1,c2)
          <-> name(c1) = name(c2) &
              (isAbstract(c1) <-> isAbstract(c2)) &
              \forall int i; (attributeDefined(c1,i) <-> attributeDefined(c2,i)) &
              \forall int i; (attributeDefined(c1,i)
                   -> name(attribute(c1,i)) = name(attribute(c2,i)))))
```

Fig. 8. Encoding of isomorphism for a whole class group

Figure 8 shows (in KeY syntax) the definition of the isomorphism-predicate for the class group containing class `Class`. Two instances of `Class` are isomorphic if their attributes have the same value and if the sequence of linked attributes are isomorphic. The sequence of linked attributes is encoded for Simplify by a function `attribute` with two arguments, the second argument encodes the position within the sequence. Two sequences of attributes are isomorphic, if they contain on the same position always isomorphic elements.

The final step is the generation of a proof obligation for each class group; the proof obligations have the same structure as the one discussed in Sect. 4.1. For

[3] However, one could easily solve this problem by adding a new display manager class `AttributeDM` to the concrete syntax definition.

our example, SIMPLIFY was again successful in discharging all proof obligations fully automatically.

5 Related Work

Our approach of defining the concrete syntax presented in Sect. 3 has many similarities with Triple-Graph-Grammars, already invented by Schürr in 1994 [7] (see also [10] for a more recent survey and a case study). The most important difference between our approach and TGGs is that our goal is merely to describe valid instances of the concrete syntax metamodel, but we are not interested in how such instances are constructed. While the main idea of defining the concrete syntax is quite similar to TGG, we are not aware of any work in the TGG area, that aims at analyzing TGG definitions as we do.

Xia and Glinz present in [11] an approach to describe the concrete syntax of their own graphical modeling language ADORA [12]. The main idea is to map the graphical representation of a language construct to a textual representation and to define the syntax finally in EBNF style. One restriction of this approach is that each graphical element must correspond to exactly one model element, and vice versa. On the other hand, Xia/Glinz were able to handle advanced features like graphical nesting in an elegant way, the constraints they give are much more concise than the corresponding invariants we could give as OCL invariants in display manager classes.

6 Conclusion and Future Work

In this paper, we have described an approach to formally define the concrete syntax of a modeling language. The formalization we propose is directly based on the primary language definition, i.e. the metamodel that encodes the abstract syntax. The big advantage of having a formalized version of the concrete syntax definition is, compared to informal syntax definitions, the possibility to analyze automatically correctness properties (cmp. Sect. 4). If the syntax definition is incorrect, our rigorous analysis is able to report an erroneous situation. For correct definitions, our approach is able to certify that erroneous situations never occur.

So far, we have encoded all proof obligations manually but we plan to automatize this step in a tool dedicated to formal concrete syntax definition. This tool should also provide a visualization of the counterexamples found by SIMPLIFY, so that the user of the tool gets feedback in the same format in which the concrete syntax is defined. Another direction of future activities is the development of an OCL axiom library that codifies the knowledge on OCL's predefined data structures. For example, the fact that for all sets `s` and elements `x` the term `s->including(x)->excluding(x)` is semantically equivalent to `s` is sometimes needed. Such axiom libraries have been developed extensively for other specification languages, e.g. Z, but – to our knowledge – not for OCL, yet. It is very likely, that SIMPLIFY will show some weaknesses in proving proof obligations, once the proof requires certain types of axioms, e.g. axioms describing sophisticated properties of sets. For this case, we plan to integrate other decision procedures or model checkers into our tool.

References

1. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, second edition, 2005.
2. Frédéric Fondement and Thomas Baar. Making metamodels aware of concrete syntax. In Alan Hartman and David Kreische, editors, *Proc. European Conference on Model Driven Architecture (ECMDA-FA)*, volume 3748 of *LNCS*, pages 190–204. Springer, 2005.
3. D. L. Detlefs, G. Nelson, and J. Saxe. Simplify: the ESC theorem prover. Technical report, DEC, 1996.
4. OMG. Unified Modeling Language: Diagram interchange version 2.0. Convenience Document ptc/05-06-04, June 2005.
5. Gennaro Costagliola, Andrea De Lucia, Sergio Orefice, and Giuseppe Polese. A classification framework to support the design of visual languages. *Journal of Visual Languages and Computing*, 13(6):573–600, 2002.
6. Fabien Rohrer and François Helg. Synchronization between display objects and representation templates in graphical language construction. Minor thesis at Software Engineering Laboratory of EPFL, 2006. Available from http://lglpc35.epfl.ch/lgl/members/fondement/projects/probxs/HelgRohrer.pdf.
7. Andy Schürr. Specification of graph translators with triple graph grammars. In *Proceedings of 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'94)*, volume 903 of *LNCS*, pages 151–163. Springer, 1995.
8. M. Pressburger. Über de vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchen, die addition als einzige operation hervortritt. *Sprawozdanie z I Kongresu Matematikow Krajow Slowcanskich Warszawa*, pages 92–101, 1929.
9. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *The KeY Book – The Road to Verified Software.* Springer, 2006. To appear.
10. A. Königs and A. Schürr. Tool Integration with Triple Graph Grammars - A Survey. In R. Heckel, editor, *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 113–150, Amsterdam, 2006. Elsevier Science Publ.
11. Yong Xia and Martin Glinz. Rigorous EBNF-based definition for a graphic modeling language. In *Proceedings of 10th Asia-Pacific Software Engineering Conference (APSEC 2003)*, pages 186–196. IEEE Computer Society Press, 2003.
12. Martin Glinz, Stefan Berner, and Stefan Joos. Object-oriented modeling with ADORA. *Information Systems*, 27(6):425–444, 2002.