Softw Syst Model (2008) 7:383–398
DOI 10.1007/s10270-008-0086-z

SPECIAL SECTION PAPER

Correctly defined concrete syntax

Thomas Baar

Received: 19 March 2007 / Revised: 8 September 2007 / Accepted: 19 September 2007 / Published online: 1 July 2008

Abstract Due to their complexity, the syntax of modern modeling languages is preferably defined in two steps. The abstract syntax identifies all modeling concepts whereas the concrete syntax should clarify how these concepts are rendered by graphical and/or textual elements. While the abstract syntax is often defined in form of a metamodel, there does not exist such standard format yet for concrete syntax definitions. The diversity of definition formats—ranging from EBNF grammars to informal text—is becoming a major obstacle for advances in modeling language engineering, including the automatic generation of editors. In this paper, we propose a uniform format for concrete syntax definitions. Our approach captures both textual and graphical model representations and even allows to assign more than one rendering to the same modeling concept. Consequently, following our approach, a model can have multiple, fully equivalent representations, but—in order to avoid ambiguities when reading a model representation—two different models should always have distinguishable representations. We call a syntax definition correct, if all well-formed models are represented in a non-ambiguous way. As the main contribution of this paper, we present a rigorous analysis technique to check the correctness of concrete syntax definitions.

Keywords Visual languages · Concrete syntax · Metamodeling · OCL · Triple-Graph-Grammars (TGGs)

Communicated by Prof. O. Nierstrasz.

T. Baar (✉)
School of Computer and Communication Sciences,
École Polytechnique Fédérale de Lausanne (EPFL),
1015 Lausanne, Switzerland
e-mail: thomas.baar@acm.org

1 Introduction

The trend to model-driven development is facing the question how modeling languages can be defined precisely in a standardized format. Metamodeling became in the last decade the prevailing technique for describing the *abstract syntax* of modeling languages: metaclasses represent all modeling concepts, metaattributes their variations, and metaassociations their relationships. In addition, well-formedness rules written as OCL invariants impose restrictions, which have to be satisfied in each well-formed instance of the abstract syntax metamodel. In the literature, well-formed instances of the abstract syntax metamodel are also known as *models* and we will stick to this established terminology also thorough this paper.

Though the metamodel capturing the abstract syntax describes precisely the structure of all possible models, the metamodel is not yet a complete definition of a modeling language. In order to be complete, a language definition needs two additional parts describing (1) how a model is rendered by textual and/or graphical elements (concrete syntax) and (2) what the intended meaning of each modeling concept is (semantics). Unfortunately, there is no commonly agreed format for these two parts, yet. This paper is only concerned about the first missing part (concrete syntax definitions) and will ignore entirely the problem of how the semantics of modeling concepts can be formally defined (interested readers are referred to [1] for a complementary approach on defining semantics).

When defining the concrete syntax, one has to prescribe—as a first step—which graphical and/or textual elements are available for the representation of model elements (e.g., boxes, lines, stickmen, text). Moreover, possible relationships between representation elements need to be specified, e.g., that a line always connects two boxes, that text can

appear either in the first compartment of a box or as a label of a line, etc. This selection of possible representation elements is realized in our approach by the definition of a *representation language* in form of a formal metamodel (cmp. Sect. 2). Fortunately, there are many modeling languages that share the same representation language what allows to reuse representation metamodels in multiple concrete syntax definitions.

Once the representation language is fixed, one can complete the concrete syntax definition by describing the relationship between model elements and representation elements. In our approach, this is done by bridging the two metamodels for abstract syntax and representation language with some additional metaclasses, metaassociations and OCL constraints (Sect. 3).

As an example, let us consider the official language definition for UML (cmp. [2]). The metamodel for the abstract syntax identifies many modeling concepts, such as `Classifier`, `State`, `Actor`, etc. A UML model is just an instance of this metamodel; otherwise stated, a set of classifiers, states, actors, etc. together with values for metaattributes and metaassociations. The UML language reference, however, describes only informally how to render the modeling concepts identified in the abstract syntax metamodel. While such an informal concrete syntax definition might have some advantages with respect to readability, there are also many compelling reasons to adopt a more formal approach when defining a modeling language including its concrete syntax:

- The metamodel capturing the abstract syntax is most often developed having a particular graphical or textual representation of the model in mind. Developing an explicit metamodel of the representation language enforces the language designer to strictly separate modeling concepts from representation concepts.
- Formal definitions are amenable to formal analysis. In Sect. 4, we present an analysis algorithm that is able to check, whether modeling and representation language are bridged correctly. Surprisingly—as our examples will illustrate—problems found by this analysis are often not due to an incorrect specification of the bridge but due to some imprecision in the definition of the modeling language itself. Consequently, defining the concrete syntax formally and conducting a correctness analysis can also be seen as a technique to *validate* a given abstract syntax metamodel and to detect its inconsistencies.
- Finally, a formal definition is less prone to ambiguities and misinterpretations.

This paper is an extended version of [3] and is composed of two main parts. In the first part, Sects. 2 and 3, we explain how the metamodeling approach can be adopted for the precise definition of presentation languages and, finally, for the formal definition of the concrete syntax of modeling languages. This part of the paper goes back to ideas first described in [4]. The paper's second part, Sect. 4, is a much enhanced version of the corresponding section in [3] and describes how the correctness of concrete syntax definitions can be rigorously analyzed and, in addition, how modern deduction tools can assist this analysis process effectively. A concrete syntax definition is called *correct*, iff it disallows two different models to be rendered by the same representation. Otherwise stated, the relationship *model-representation* is, in mathematical terms, a function from (the set of) all possible representations to (the set of) all possible models. Note, however, that for the opposite direction the rendering relationship does not need to be functional. Our syntax definitions allow the same model to be represented by more than one representation. This is sometimes called *presentation option* in the literature. For example, in UML, one can mark a class to be abstract either by adding a stereotype *abstract* to the class representation or by setting the class name into an italic font [2].

At the end of the paper, Sect. 5 gives an overview on related work while Sect. 6 summarizes the most important achievements of our approach.

2 Representation languages

In this section, we give some background information on the definition of graphical and textual representation languages in form of a metamodel.

2.1 Graphical languages

Some modeling languages use for the representation of models graphical elements, such as boxes, circles, lines, stickmen, etc. Graphical elements are also called *visual objects* in the literature [5], since they can be easily described as objects whose state is determined by values for certain attributes, such as *shape*, *lineColor*, *backgroundColor*, *attachRegion*, etc. An ensemble of visual objects is a syntactically correct *sentence of a graphical language* $\mathcal{L}$ iff all well-formedness rules of language $\mathcal{L}$ are met. A typical example for a well-formedness rule is the restriction that visual objects of type *Edge* must always connect two other visual objects. More technically, that the start- and endpoint of an edge is placed inside the attach regions of the connected visual objects. A syntactically correct sentence of a graphical language is also called *diagram*.

For all non-trivial graphical languages, it has advantages to identify different classes of visual objects because not all attributes are relevant for each object. For example, a visual object of type *Edge* does not need a value for the attribute

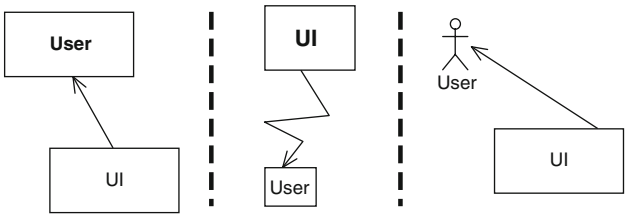


Fig. 1 Three diagrams—when read as representations of class diagrams, the first two diagrams should not be distinguishable

backgroundColor. Once the classes of visual objects together with their attributes are identified, many well-formedness rules of the graphical language can be straightforwardly expressed using the associations between these classes. The above given restriction for objects of type *Edge* to connect exactly two other visual objects is, for instance, best expressed by two associations from class `Edge` to a class, say, `LabeledNode` (which represents the connected visual objects) with multiplicity 1 at the latter class (cmp. upper part of Fig. 2).

Note that the formalization of graphical elements as visual objects, which are grouped in classes sharing the same attributes, is nothing but a metamodel of a graphical language. In this case, a diagram can be seen just as an instance of the graphical language metamodel.

The three diagrams given in Fig. 1 illustrate the sketched metamodeling approach for defining representation languages. At a first glance, the three diagrams consist of labeled rectangles and a stickmen, which are connected by a direct line or, in case of the middle diagram, by a polyline. The rectangles have different size and positions within the diagrams and also their labels differ in terms of font style and font size. To summarize, the three diagrams look all quite different.

If the diagrams, however, are *interpreted* as UML class diagrams, then the first two diagrams coincide. It is worth to investigate the process how humans *interpret* an ensemble of graphical objects as a UML class diagram a little bit further. Humans learned by reading the informal concrete syntax definition of UML given in [2] that the size of a class box does not matter for UML class diagrams, that the font style and font size of a label does not matter (except for abstract classes), that the position of rectangles in the diagram does not matter (as long as one does not contain the other), that the usage of direct line versus polyline does not matter, etc. The concrete syntax definition is the place to specify which of all possible attributes of visual objects are *relevant for the rendered modeling language*.

Figure 2 shows one possible metamodel for the graphical representation language used in Fig. 1. The chosen metamodel consists of only such attributes that are considered to be relevant when interpreting the three diagrams from Fig. 1

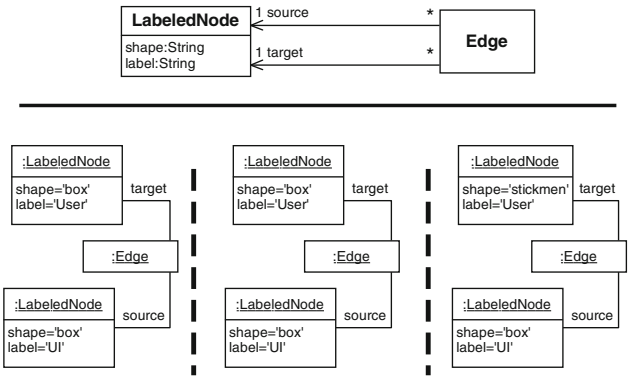


Fig. 2 A graphical language definition and corresponding representation of diagrams given in Fig. 1

as UML class diagrams. In the lower part of Fig. 2, the three diagrams are given as instances of the representation language metamodel. The first two diagrams coincide, indeed, but they differ from the third diagram.

The formalization of diagrams as instances of a metamodel is not a new technique and actually every graphical editor has internal data structures, which resemble such a metamodel in one way or the other. A warmly recommended introduction to graphical languages is given in [5]. Here, Costagliola et al. propose a classification of graphical languages (*geometric-based, connection-based, hybrid*) and derive from this classification a format for their concise and precise definition (see also [6] for an implementation strategy). A core technique is to substitute absolute layout information (such as *position*, *dimension*) by relative ones, called *spatial relationships*. When defining a metamodel for a graphical language, one has to identify—in a first step—besides relevant attributes of visual objects also all relevant spatial relationships. For example, UML diagrams take two spatial relationships into account: *graphical nesting* (class diagrams, state diagrams) and the *NORTH-TO* relationship (sequence diagrams).

The OMG standard for *Diagram Interchange* (*DI*) [7] proposes a metamodel for a graphical representation language, which is expressive enough for any kind of UML diagrams (and for many domain-specific languages as well). We could have taken *DI* as a basis for all our examples and each of the results obtained in this paper would also apply in such a setting. Nevertheless, we decided to use a home-brewed version of a graphical representation language instead of *DI* in order to illustrate the independence of our approach from the chosen representation language and, also, because we did not want to bother the reader with the technical complexity of *DI*.

2.2 Textual languages

A *sentence of a textual language* is a sequence of strings, which are separated by whitespace symbols, such as *newline*,

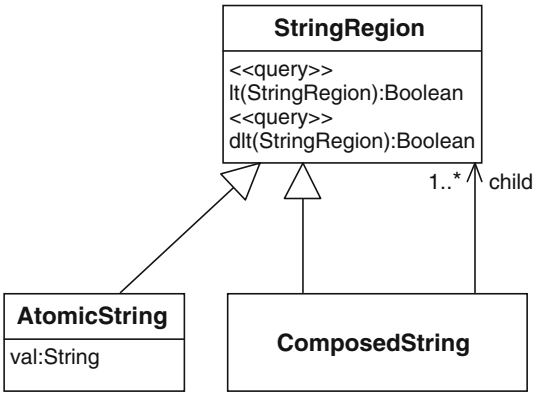


Fig. 3 Simple metamodel of a textual representation language

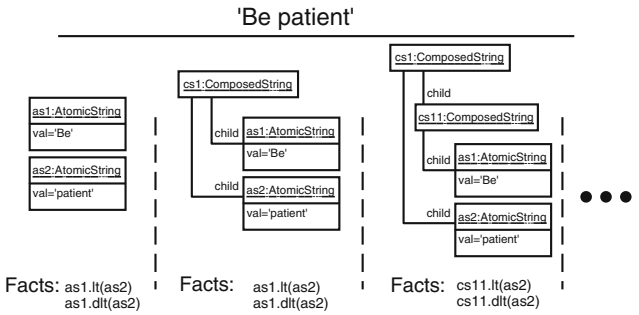


Fig. 4 Some possible representations of text 'Be patient'

tabulator, and *space*. A sequence of strings can be equivalently described by (1) saying which strings occur in the sequence and how often they occur, and by (2) determining the ordering of string occurrences using the *LEFT-TO* spatial relationship.

Based on these observations, a minimal metamodel for a textual language could consist of just one class `AtomicString`, which has an attribute `val` for the represented text and a query `lt(AtomicString):Boolean` for encoding the *LEFT-TO* relationship. Furthermore, one had to axiomatize the *LEFT-TO* relationship as a total order on `AtomicString`.

For our purpose—the definition of a concrete syntax—it is however useful to add to this minimal metamodel a metaclass for the composition of consecutive strings (`ComposedString`) and to declare a spatial relationship *DIRECT-LEFT-TO* (`dlt`), which is derived from *LEFT-TO*. The resulting metamodel is shown in Fig. 3 and augmented by a number of OCL invariants shown in Listing 1, which axiomatize the spatial relationships *LEFT-TO* and *DIRECT-LEFT-TO*.

Listing 1 Axiomatization of LEFT-TO and DIRECT-LEFT-TO

```
context StringRegion
inv irrefl_lt: not self.lt(self)
inv trans_lt: StringRegion.allInstances()->forAll
    (y,z|(self.lt(y) and y.lt(z))implies self.lt(z))
inv antisymm_lt: StringRegion.allInstances()->
     forAll(y|self.lt(y) implies not y.lt(self))
-- dlt is a derived relationship from lt
inv definition_dlt: StringRegion.allInstances()->forAll(y|
     self.dlt(y) =
       not StringRegion.allInstances()->exists(z|
         self.lt(z) and z.lt(y)))
```

Figure 4 shows three different instantiations of the textual language metamodel, which all represent the same text *Be patient*. Note that the same text can be represented in many different ways. The question, how these representations can be created or derived from a given text, is, however, out of scope of this paper.

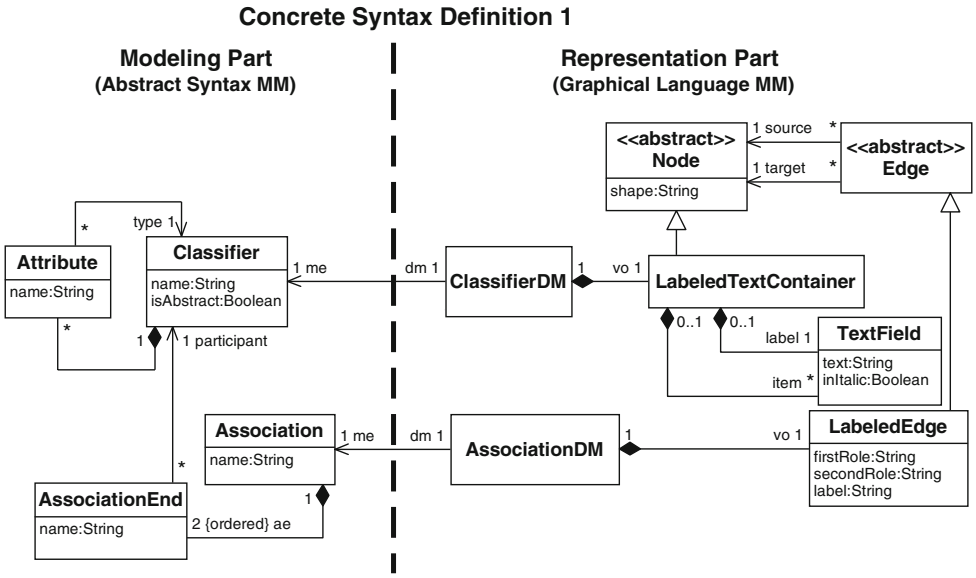
3 Concrete syntax definition

In the previous section, we have outlined how graphical/textual representation languages can be formalized in form of a metamodel; we will now answer the question how instances of an abstract syntax metamodel can be rendered in such representation languages. The missing part is, informally speaking, the bridge from the metamodel of the abstract syntax to the metamodel of the representation language. We will illustrate our approach using a simplified version of UML class diagrams that features three kinds of model elements: classes (classifiers), attributes, and associations.

3.1 Graphical languages

Figure 5 presents our definition of the graphical concrete syntax of UML class diagrams. The left part contains the metamodel of the modeling language; here, UML class models: each (instance of) `Classifier` owns a set of `Attributes` (see composition between `Classifier` and `Attribute`) and each `Attribute` in turn refers to a `Classifier` as its type. Each `Association` owns two `AssociationEnds`, each of which refer to exactly one `Classifier` as its participant. The right part of Fig. 5 contains the metamodel of the representation language, which classifies possible graphical objects in class diagrams (e.g., `LabeledTextContainer`, `LabeledEdge`) and identifies relevant attributes.

The two classes `ClassifierDM` and `AssociationDM` are so-called *display manager classes* (the name of these classes has, by convention, always the suffix `DM`). Display manager classes are actually the "piers" for the bridge from the modeling to the representation language. A display manager class is always connected via an association with multiplicity 1-1 to one modeling class (i.e., a class from the modeling language metamodel). The purpose of display manager classes is to manage the rendering of the instances of the referred modeling class. By convention, we always use `me` (for **m**odel **e**lement) and `dm` (for **d**isplay **m**anager) as role names on the association between display manager class and



managed class. Moreover, display manager classes are connected via composition with multiplicity 1 and rolename `vo` (for **v**isual **o**bject) to a representation class (i.e., a class from the representation language metamodel).

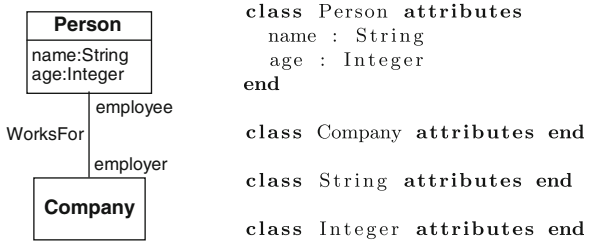
The bridge between modeling and representation language is specified by OCL invariants, which are normally attached to the display manager classes. These invariants formalize conditions for synchronizing the state of modeling elements with the state of representation objects (which realize the rendering of modeling elements).

Listing 2 Invariants of Concrete Syntax Definition 1 (graphical rendering)

```
context ClassifierDM
inv shape: self.vo.shape='box'
inv label:  self.me.name = self.vo.label.text and
            self.me.isAbstract = self.vo.label.inItalic
inv attributes: self.vo.item.text->asSet() =
     self.me.attribute->collect(att|
       att.name.concat(':').concat(att.type.name))->asSet()

context AssociationDM
inv roles: self.vo.firstRole = self.me.ae->at(1).name and
     self.vo.secondRole = self.me.ae->at(2).name and
     self.vo.label = self.me.name
inv connections: self.vo.source = self.me.ae
    ->at(1).participant.dm.vo and
     self.vo.target = self.me.ae->at(2).participant.dm.vo
```

For the example shown in Fig. 5, the necessary OCL constraints are given in Listing 2. The first constraint `shape` stipulates that a `Classifier` is always represented by a box-shaped node. The next two constraints `label` and `attributes` formulate the synchronization conditions for `Classifiers` and their representations (instances of `LabeledTextContainer` and `TextField`). Furthermore, from the invariants for `AssociationDM`, one can conclude that each `Association` is rendered by a `LabeledEdge`, whose values for `firstRole`, `secondRole` and `label` correspond to the name of the association and to the name of its two association ends. Furthermore, the edge connects those two nodes that render the classifiers to which the two association ends refer as participants.



(a) Graphical representation **(b)** Textual representation

Fig. 6 Example of graphical class diagram and its textual representation as USE input

3.2 Textual languages

As an example of a purely textual representation of class diagrams, we investigate in this subsection the input format of the USE tool [8]. The USE tool is a popular and often cited OCL tool, which allows its users to enter both OCL constraints and the underlying class diagram in a purely textual format. A tiny example is given in Fig. 6.

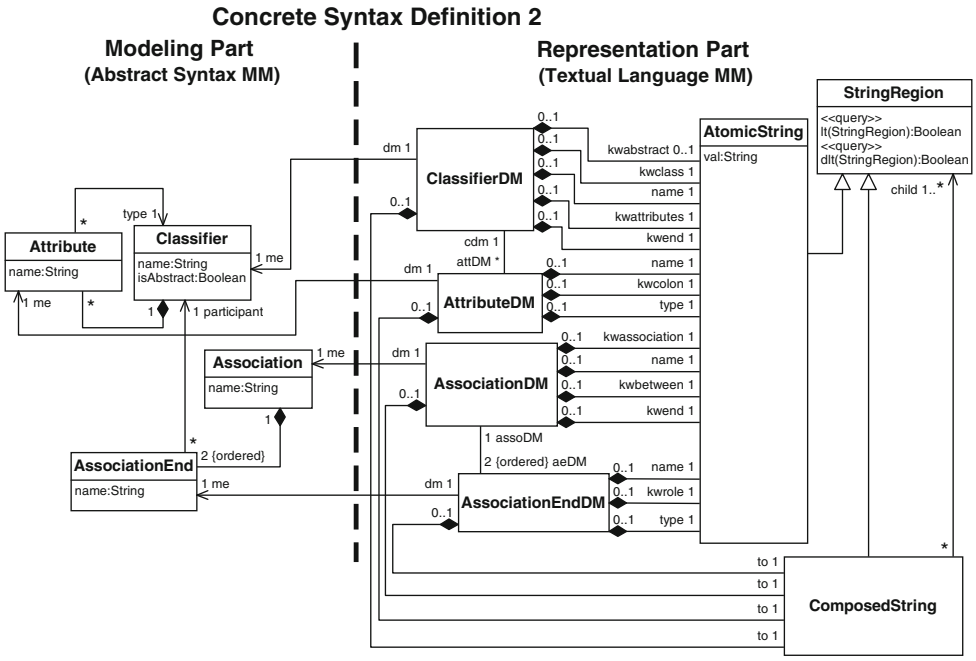


Fig. 7 Bridging the metamodels describing abstract syntax and textual representation language

The expected format of the textual input files for class diagrams was specified by the USE developers by a traditional EBNF grammar[1] that looks as follows:

```
classDefinition ::=
  [ ''abstract'' ] ''class'' id
    ''attributes'' { attributeDefinition }
  ''end''

attributeDefinition ::=
  id '':'' id

associationDefinition ::=
  ''association''
  id ''between''
  associationEnd associationEnd
  ''end''

associationEnd ::=
  id [ ''role'' id ]
```

Grammar-based language definitions suffer from the general problem to encode merely the concrete syntax tree but to leave the relationship to the abstract syntax open (see also Muller et al. in [9]). For instance, the non-terminal `id` appears in clause `classDefinition` but it is nowhere stated what `id` actually represents. The information that `id` is a placeholder for a class name has to be given elsewhere.

Figure 7 shows an alternative concrete syntax definition for the USE input files, which follows our approach. This definition resembles literally the above given USE grammar but specifies in addition also formally the bridge to the abstract syntax. It uses the simple metamodel for textual representation languages shown in Fig. 3 and has—compared to the definition of a graphical representation—more display manager classes (one for each metaclass of the modeling language metamodel). Each display manager class encodes one clause in the USE grammar.

All display manager classes have a directed association with multiplicity 1 and role `to` (for **t**ext **o**bject) to class `ComposedString`. The referred `ComposedString` represents a region of strings and contains as children exactly the elements that appear at the right hand side of the corresponding grammar rule. The mapping to the abstract syntax as well as the spatial relationships between the strings are encoded as OCL invariants. For the purpose of illustration, the constraints for `ClassifierDM` are given in Listing 3. The constraints for each display manager class can be classified into four groups:

keyword For each keyword occurring in the right hand side of the corresponding grammar rule, the display manager class has a directed association to `AtomicString` with a suitable role name (e.g. `kwclass` in `ClassifierDM`). Keyword constraints ensure that the referred instance of `AtomicString` has the expected value.

mapping Mapping constraints establish the bridge between modeling and representation elements.

child Child constraints declare all children of the referred textual object.

[1] The grammar presented here is a simplified version and has dropped all elements not needed for our running example, e.g. operation declarations, specialization between classes, different kinds of associations (aggregation, composition), etc.

ordering Ordering constraints define the ordering between the children.

Listing 3 Selected Invariants of Concrete Syntax Definition 2

```
context ClassifierDM
— value of keywords
inv keyword:
    (self.kwabstract->notEmpty() implies
       self.kwabstract.val='abstract') and
    self.kwclass.val='class' and
    self.kwattributes.val='attributes' and
    self.kwend.val='end'
— mapping to model elements
inv mapping:
    self.name.val=self.me.name
— children of self.to
inv child:
    let mandatoryElements:Set(StringRegion) =
      self.attDM.to
      ->union(Set{self.kwclass,self.name,
            self.kwattributes,self.kwend})
    in
      self.to.child =
        if self.kwabstract->isEmpty()
        then mandatoryElements
        else mandatoryElements->including(self.kwabstract)
        endif
— ordering of children
inv ordering:
    (self.kwabstract->notEmpty()
          implies self.kwabstract.dlt(self.kwclass)) and
    self.kwclass.dlt(self.name) and
    self.name.dlt(self.kwattributes) and
    self.attribute.to->forAll(a|
          self.kwattributes.lt(a) and
          a.lt(self.kwend))
```

3.3 Summary

In our approach, concrete syntax definitions

- specify in a purely declarative way all possible cases how a model (an instance of the abstract syntax metamodel) can be bridged to its graphical/textual representation (instance of the representation language metamodel). The bridge is formally encoded by OCL invariants, which are usually attached to display manager classes. Note that these invariants can be arbitrary boolean OCL expressions, what makes our approach very flexible and allows, for example, to define *presentation options*, i.e. the same model can be shown by different, i.e. non-isomorphic, representations.
- do not require to define a display manager class for each class of the abstract syntax metamodel. In the above given concrete syntax definition for graphical representations of class diagrams (see Fig. 5), display manager classes are defined only for the metaclasses `Classifier` and `Association`. The classes `Attribute` and `AssociationEnd` do not need their own display manager class, because the rendering of these classes is already captured by `ClassifierDM` and `AssociationDM`.

4 Analysis of concrete syntax definitions

As already mentioned in the introduction, concrete syntax definitions can be incorrect. Correctness means in our context that each instance of the representation metamodel that is well-formed and that satisfies the bridge constraints attached to the display manager classes corresponds to exactly one instance of the abstract syntax metamodel. Otherwise stated, it must be always possible to *construct* the model from the given representation of this model in a unique way.

This section presents two techniques for checking the correctness of concrete syntax definitions rigorously. Both techniques are—from the mathematical point of view—equivalent since they both allow the user to prove or to disprove the correctness of a given concrete syntax definition. The second technique presented in Sect. 4.3, however, is more amenable to automatization since it relies on the generation of proof obligations, which, in principle, can be discarded by automatic deduction tools.

Before digging into details, we shortly summarize how a language designer is rewarded for the effort of conducting a rigorous correctness analysis:

- By analyzing the concrete syntax definition, one also cross-checks the representation language against the modeling language. The representation language must be at least as expressive as the modeling language (recall that all information about a model should be obtained from the given representation of the model). Our analysis algorithm will make explicit all side-conditions that must hold to ensure a unique construction of a model from a given representation. As we will see later, this especially means that our analysis algorithm can detect the incompleteness of one metamodel with respect to the other metamodel (i.e. some well-formedness rules have been forgotten in the first metamodel). Thus, the correctness analysis can also be seen as a validation technique on the completeness of a given metamodel.
- The analysis can be done as a paper-and-pencil work once the metamodels for modeling and representation language are available. Consequently, possible inconsistencies in the metamodels can be detected prior to the (cost-intensive) development of tools (e.g. editors) for the defined modeling language. After the analysis, all subsequent activities can be done with higher confidence that the defined metamodels really reflect the intentions of the language designer.

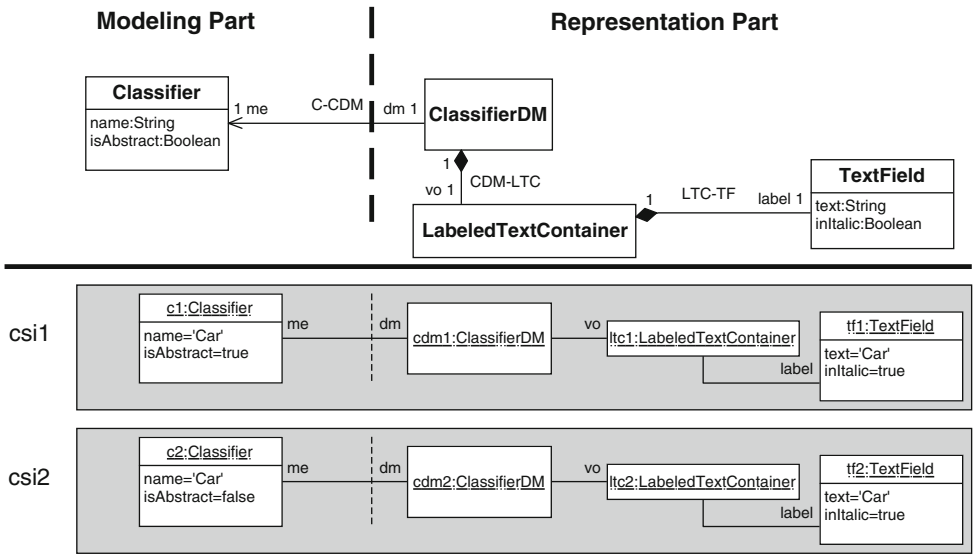


Fig. 8 Example for an incorrect syntax definition and counterexample

To summarize, correctness analysis is not only a verification activity but also a validation activity for the involved metamodels.

4.1 Correctness criterion

We start with an example of an incorrect syntax definition and derive from this example afterwards a general criterion for the correctness of concrete syntax definitions.

4.1.1 Illustrating example

The upper part of Fig. 8 is a simplified version of the running example shown in Fig. 5 for the definition of a graphical concrete syntax (note that for technical reasons, however, we added association names to the metamodel shown in Fig. 8). Suppose, only the following invariant were attached to the concrete syntax definition:

context ClassifierDM **inv** nameMapping:
`self.me.name=self.vo.label.text`

The lower part of Fig. 8 shows two instantiations *csi1*, *csi2* of the concrete syntax metamodel, which conform to all multiplicity constraints made in the metamodel and to the invariant attached to `ClassifierDM`. These two instantiations witness an error in the concrete syntax definition: It is possible that two ≪ different≫ models (note that objects *c1*, *c2* of class `Classifier` have different values for attribute `isAbstract`) are represented by the ≪same≫ representation. Otherwise stated, the representation part of *csi1* (which coincides with the representation part of *csi2*) does not define a unique model. These erroneous instantiations *csi1*, *csi2* are possible because the invariant `nameMapping` attached to `ClassifierDM` stipulates how attribute `name` in class `Classifier` is represented, but nothing is said on the representation of attribute `isAbstract`.

The correctness analysis presented below is able to create automatically witness scenarios as the one shown in the lower part of Fig. 8. If witness scenarios do not exist, the correctness analysis verifies systematically their absence. If the analysis could find some errors in a syntax definitions, the found errors can be fixed by the language designer in two ways: (1) strengthening the mapping constraints attached to the display manager classes or (2) adding new well-formedness rules to the abstract syntax metamodel.

4.1.2 Mathematical definitions

When arguing for the incorrectness of the above given syntax definition, we have been quite vague when saying that two models are ≪the same≫ or that they are ≪different≫. What we actually meant, was, that two models are—when seen as instances of their metamodel—isomorphic or non-isomorphic graphs. For a better understanding of the remainder of this section, we now summarize here the relevant mathematical foundation of graph-isomorphism.

Definition 1 (Isomorphic sets) Two sets A, B are *isomorphic* iff there exists a function m from A to B (Notation: $m : A \longrightarrow B$), which is bijective.

A function $m : A \longrightarrow B$ is called *bijective* iff m is

- total (for each $a \in A$ exists exactly one $b \in B$ such that $m(a) = b$)
- injective (two different arguments result in two different function values: $a1 \neq a2$ implies $m(a1) \neq m(a2)$)

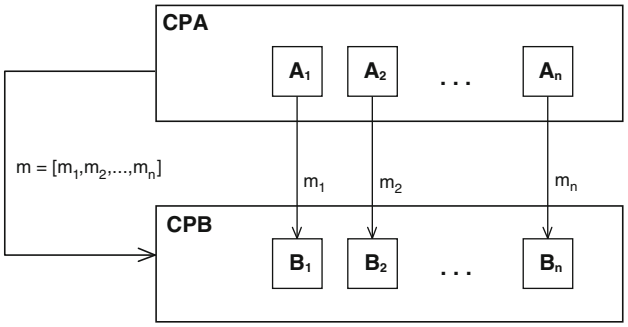


Fig. 9 Isomorphic Cartesian products

– surjective (for each $b \in B$ there exists one $a \in A$ such that $m(a) = b$)

Recall that two isomorphic sets A, B have always the same cardinality (i.e., the same number of elements) and that there is a bijective reverse function $m^{-1} : B \longrightarrow A$ such that $m^{-1}(m(a)) = a$.

Fact 1 (Isomorphic Cartesian products) *Let* $CPA = [A_1, \ldots, A_n]$ *and* $CPB = [B_1, \ldots, B_n]$ *be two Cartesian Products of sets* A_i, B_i *for* $i = 1, \ldots, n$.

If A_i *is isomorphic to* B_i *for all* $i = 1, \ldots, n$ *then CPA is isomorphic to CPB.*

Figure 9 illustrates the situation described by Fact 1. Note that the bijection $m : CPA \longrightarrow CPB$ can be simply constructed by composing all existing bijections $m_i : A_i \longrightarrow B_i$ (i.e. $m = [m_1, \ldots, m_n]$).

Fact 2 (Isomorphic structures) *Let* $STRA = [A_1, \ldots, A_n, fa_1, \ldots, fa_l]$ *and* $STRB = [B_1, \ldots, B_n, fb_1, \ldots, fb_l]$ *be two structures* (*Cartesian Products of sets and functions*), *where* $fa_k : A_{k1} \longrightarrow A_{k2}, fb_k : B_{k1} \longrightarrow B_{k2}$ *for* $k = 1..l$ *are functions between sets of the same structure* (Fig. 10).

If A_i *is isomorphic to* B_i *for all* $i = 1, \ldots, n$ *and the functions* fa_k, fb_k *commute over the bijection* $m = [m_1, \ldots, m_n]$ *with* $m_i : A_i \longrightarrow B_i$ *then STRA is isomorphic to STRB.*

The functions $fa_k : A_{k1} \longrightarrow A_{k2}$ *and* $fb_k : B_{k1} \longrightarrow B_{k2}$ commute *over the bijection* $m = [m_1, \ldots, m_n]$ *iff for all* $a \in A_{k1}$ *the following holds*:

$m_{k2}(fa_k(a)) = fb_k(m_{k1}(a))$

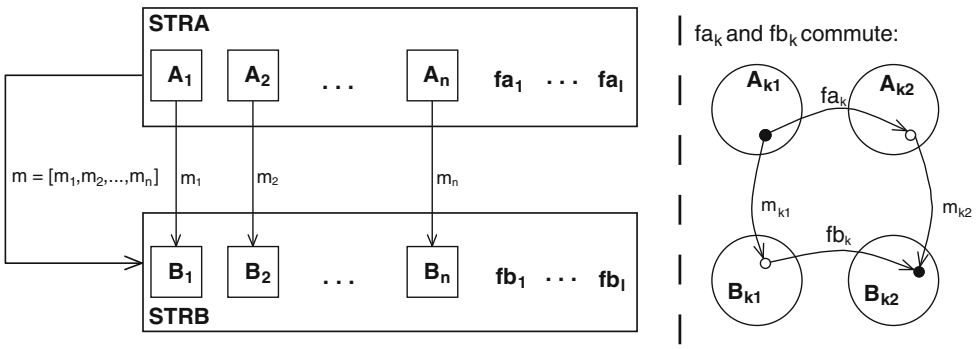
Based on this mathematical foundation, we are now almost ready to precisely argue, why for the counter example given in the lower part of Fig. 8 the two representations are isomorphic but the two represented models are not. The only step left open is to encode the instantiations of metamodels as mathematical structures (i.e., Cartesian Products of sets and functions).

An instantiation of a metamodel has three parts: (1) the set of existing objects for each non-abstract metaclass, (2) the assignment of values to attributes for each object, (3) the set of existing links for each metaassociation whereas each link connects exactly two objects. We will encode part (1) and part (2) straightforwardly, but part (3) (set of links) needs special attention since a naive encoding would result into a very clumsy data structure.

The set of links can be encoded for each metaassociation *as* by a relation $r_{as} : Obj_{\text{SourceClass}} \longleftrightarrow Obj_{\text{TargetClass}}$, where $(o_1, o_2) \in r_{as}$ iff there exists a link for metaassociation *as* between o_1 and o_2. This encoding, however, requires to assign a direction to metaassociation *as* (i.e., determine, which of the two classes participating in *as* is the SourceClass and which is the TargetClass). In order to indicate this direction in our examples, we slightly abuse UML's navigation arrows (note that in Figs. 5 and 7 all plain associations are directed). If the association is a composition, we can omit the navigation arrow since the container class should always be read as the source class and the contained class is always read as the target class.

As a last optimization of the encoding, we will use instead of the relation r_{as} its functional counterpart $f_{as} : Obj_{SourceClass} \longrightarrow TargetType$, which yields for a given object o_1 of the source class all objects that are connected with o_1 via a link for *as*. The result type *TargetType* of f_{as} depends on the multiplicity and ordering attached to

Fig. 10 Isomorphic structures



association *as* and can, in our examples, be $Obj_{TargetClass}$, `Set`($Obj_{TargetClass}$) or `OrderedSet`($Obj_{TargetClass}$).

Definition 2 (Concrete syntax instantiation) An *instantiation* of a concrete syntax metamodel is a structure
$csi = [Obj_{C_1}, \ldots, Obj_{C_k}, DATA, f_{ae_1}, \ldots, f_{ae_m}, att_{a_1}, \ldots, att_{a_l}]$ where

- Obj_{C_i} is a set denoting the existing objects of class C_i, where C_i is a metaclass of the metamodel
- *DATA* denotes the semantic domain of all predefined types, such as String, Integer, Boolean, etc.
- f_{ae_i} is a function encoding the links stemming from the association, for which ae_i is the target association end
- att_{a_i} is a function encoding values of attribute a_i for the objects that are instances of the owner class of a_i

We call the part of *csi* that instantiates the representation part of the concrete syntax metamodel also *representation part of csi* and the remaining part the *modeling part of csi*.

Example We consider the syntax definition in the upper part of Fig. 8. Instances of this definition are encoded as a structure

$$csi = [\ Obj_C, Obj_{CDM}, Obj_{LTC}, Obj_{TF}, DATA, f_{me}, f_{vo}, f_{label}, att_{\text{name}}, att_{\text{isAbstract}}, att_{\text{text}}, att_{\text{inItalic}}]$$

where

- $Obj_C, Obj_{CDM}, Obj_{LTC}, Obj_{TF}$ denote the sets of existing objects of metaclasses `Classifier`, `Classifier-DM`, `LabeledTextContainer`, `TextField`, respectively
- $f_{me}, f_{vo}, f_{label}$ are functions encoding the links stemming from metaassociations `C-CDM`, `CDM-LTC`, `LTC-TF`, respectively. Due to the multiplicities, the functions have the following signatures:
  $f_{me} : Obj_{CDM} \longrightarrow Obj_C$,
  $f_{vo} : Obj_{CDM} \longrightarrow Obj_{LTC}$,
  $f_{label} : Obj_{LTC} \longrightarrow Obj_{TF}$
- $att_{name} : Obj_C \longrightarrow DATA$ is a function encoding values of metaattribute `name` for objects of metaclass `Classifier`;
  $att_{isAbstract}, att_{\text{text}}, att_{inItalic}$ are analogous functions for the remaining metaattributes

The instantiations *csi1* and *csi2* shown in the lower part of Fig. 8 can now be encoded as structures as follows:

$$csi1 = [\ \{c1\}, \{cdm1\}, \{ltc1\}, \{tf1\}, DATA, \{(cdm1, c1)\}, \{(cdm1, ltc1)\}, \{(ltc1, tf1)\}, \{(c1,' Car')\}, \{(c1, true)\}, \{(tf1,' Car')\}, \{(tf1, true)\}]$$

$$csi2 = [\ \{c2\}, \{cdm2\}, \{ltc2\}, \{tf2\}, DATA, \{(cdm2, c2)\}, \{(cdm2, ltc2)\}, \{(ltc2, tf2)\}, \{(c2,' Car')\}, \{(c2, false)\}, \{(tf2,' Car')\}, \{(tf2, true)\}]$$

The structures *csi1* and *csi2* are not isomorphic because the only possible bijection between the object sets would be $\{c1 \mapsto c2, cdm1 \mapsto cdm2, ltc1 \mapsto ltc2, tf1 \mapsto tf2\}$. Furthermore, the elements from *DATA* are mapped by the identity function (what will always be the case in the rest of this paper). However, for this bijection, the function $att_{\text{isAbstract}}$ does not commute.

After these preparations, we are now able to define the correctness of concrete syntax definitions formally.

Definition 3 (Correctness criterion for concrete syntax definitions) Let CSMM be a concrete syntax definition given in form of a metamodel (cmp. Fig. 5) and csi1, csi2 be two arbitrary but well-formed instantiations of CSMM.

We call the concrete syntax definition CSMM *correct* if and only if the following holds:

Whenever the representation part of csi1 is isomorphic to the representation part of csi2 then csi1 must be isomorphic to csi2.

Figure 11 illustrates the situation described in Definition 3. In order to prove the correctness of *CSMM*, one has to prove that the modeling part of *csi1*/*csi2* are isomorphic. The proof is done by extending the existing bijection *m_r* between the representation parts to the bijection *m_m* between the modeling parts of *csi1*/*csi2*.

4.2 Proving correctness: Technique 1

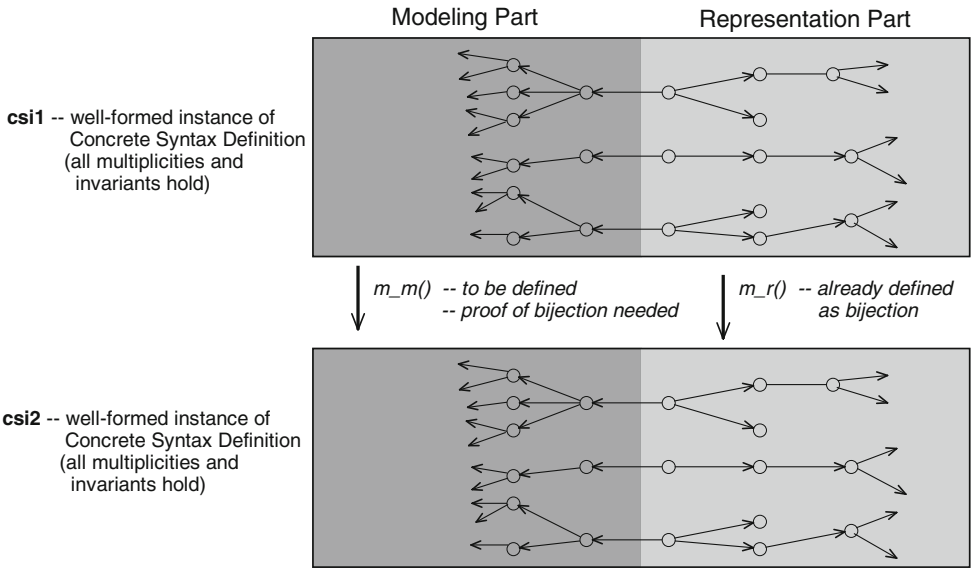
A correctness proof using this technique consists of an explicit definition for the bijection between *csi1* and *csi2*. In order to illustrate the situation, we consider a proof attempt for the correctness of the concrete syntax definition given in Sect. 4.1.1 (upper part of Fig. 8 plus constraint `name Mapping`).

Let $csi1 = [Obj^1_C, Obj^1_{CDM}, \ldots]$ and $csi2 = [Obj^2_C, Obj^2_{CDM}, \ldots]$ be given as two instantiations of the concrete syntax metamodel and *m_r* be a bijection for their representation part. We define: $m\_m : Obj^1_C \longrightarrow Obj^2_C$ as
$m\_m(c1) := f^2_{me}(m\_r((f^1_{me})^{-1}(c1)))$ for all $c1 \in Obj^1_C$

It remains to be shown that *m_m* is a bijection between Obj^1_C and Obj^2_C and that the functions belonging to the modeling part, f_{me}, att_{name} and $att_{\text{isAbstract}}$, commute over *m_m*:

- The bijection property of *m_m* follows immediately from the fact that *m_m* is defined as the composition of three bijections: $(f^1_{me})^{-1}$, *m_r*, f^2_{me}. Note that f^1_{me}, $(f^1_{me})^{-1}$ are both bijections due to the multiplicity 1 on both ends of association `C-CDM`.
- f_{me} commutes iff $m\_m(f^1_{me}(cdm1)) = f^2_{me}(m\_r(cdm1))$
  From the definition of *m_m* we obtain
  $m\_m(f^1_{me}(cdm1)) = f^2_{me}(m\_r((f^1_{me})^{-1}(f^1_{me}(cdm1))))$ what can be simplified to the conjecture since, trivially, $(f^1_{me})^{-1}(f^1_{me}(cdm1)) = cdm1$ holds.

Fig. 11 Correctness criterion



- att_{name} commutes iff $m\_m(att^1_{\text{name}}(c1)) = att^2_{\text{name}}(m\_m(c1))$
  Due to invariant `nameMapping`, we know that $att^1_{\text{name}}(c1) = att^1_{\text{text}}(f^1_{label}(f^1_{vo}((f^1_{me})^{-1}(c1))))$ and an analogous equation holds for $att^2_{\text{name}}(c1)$ Together with $m\_m(d) = d$ for all $d \in DATA$, this implies the conjecture since f^{me} commutes over m_m and $att_{\text{text}}, f_{vo}, f_{label}$ commute over m_r.
- $att_{\text{isAbstract}}$ commutes iff $m\_m(att^1_{\text{isAbstract}}(c1)) = att^2_{\text{isAbstract}}(m\_m(c1))$
  Here, the proof gets stuck since a similar invariant argument as in the case of att_{name} is missing. The problem could be resolved by adding a new invariant either to the display manager class, e.g.

  ```
  context ClassifierDM inv isAbstractMapping:
      self.me.isAbstract=self.vo.label.inItalic
  ```

  or also directly to the modeling part of *CSMM* (what, however, changes the modeling language), e.g.

  ```
  context Classifier inv isAbstractValue:
      self.me.isAbstract=false
  ```

  In this case, the remaining proof would be analogous to that for att_{name}.

Notation The usage of the strict mathematical syntax when defining the mapping (e.g., $m\_m(c1) := f^2_{me}(m\_r((f^1_{me})^{-1}(c1)))$ for all $c1 \in Obj^1_C$) or the usage of strict mathematical syntax for arguments in the proof (e.g., $att^1_{\text{name}}(c1) = att^1_{\text{text}}(f^1_{\text{label}}(f^1_{vo}(\ (f^1_{me})^{-1}(c1)))))$ can become — as the reader might have recognized—quite clumsy. Thus, we will use in the remainder of the paper a more OCL-like notation, which is as expressive as the strict mathematical one, but more readable. Instead of $f^1_{\text{assoend}}(x)$ or $f^2_{\text{assoend}}(x)$ we can write the navigation expression *x.assoend*, because from the type of x we can always conclude which of the two cases is meant. Similarly, $att^i_{\text{attrib}}(x)$ is written as *x.attrib* and $m\_r(x)$ as *x.m_r*.

4.3 Proving correctness: Technique 2

The explicit definition of *m_m* for all metaclasses in the modeling part of the syntax definition and proving its bijection property is not always as easy as in the above given example and can require substantial mental efforts from the language designer.

We describe now a variant of Technique 1, which allows a more straightforward definition of *m_m*. This Technique 2 allows an automatic generation of proof obligations (formulated in OCL), which can be discarded by an OCL prover. According to our experience, a large number of proof obligations can be discarded fully automatically. We will describe in Sect. 4.4 more in detail how Technique 2 can be enhanced with automated deduction technology.

Technique 2 is formulated in form of a two-step algorithm. The correctness of this algorithm, however, depends on the hierarchical structure of the concrete syntax definition it is applied on. More precisely, Technique 2 depends on the assumption that the modeling part of the concrete syntax definition can be partitioned, so that each partition is a tree of classes (the nodes of the tree are classes and the edges are compositions) and the root of the tree is connected via an 1-1 association to a display manager class.

4.3.1 The algorithm

The starting point is as in Technique 1: There are given two structures *csi1*, *csi2* and a bijection *m_r* between their representation parts.

Step 1 For each display manager class *DMC* and for each $dmc1 \in Obj^1_{DMC}$ assign $dmc1.me.m\_m := dmc1.m\_r.me$
This assignment defines a bijection between Obj^1_C and Obj^2_C if *C* is the class, for which *DMC* is the display manager (i.e., *C* and *DMC* are connected by an 1-1 association with role names `me` and `dm`). The bijection property of *m_m* immediately follows from the fact, that *DMC* and *C* are connected by an 1-1 association. What remains to be shown, is, that the assigned objects have the same attribute values and that outgoing links (except those stemming from a composition, these are handled separately in the next step) point to objects, which are already assigned by *m_m*/*m_r*.
PO 1 (*attributes commute*):
$dmc1.m\_r = dmc2 \text{ implies } dmc1.me.att = dmc2.me.att$
for all attributes *att* that are visible in class *C*. Recall that an attribute is visible in a class if it is declared in this class or in one of its parent classes.
PO 2 (*outgoing associations commute*):
$dmc1.m\_r = dmc2 \text{ implies } dmc1.me.ae.m\_m = dmc2.me.ae$
for all association ends *ae* of *C*'s outgoing associations (except compositions).

Step 2 For each class *C* in the modeling part do:
For each outgoing composition with association end *ae* do:
Let *T* be the type of expression *c1.ae* if *ae* had multiplicity 1. Define a selection function $sel : Set(T), T \longrightarrow T$ that gives back an element of the first argument. Then, assign
$elem1.m\_m := c1.m\_m.ae\text{->}sel(elem1)$ for each $elem1 \in c1.ae$ If association end *ae* has multiplicity 1, then *sel* must give back the single element in *c1.m_m.ae*
PO 1 (*well-definedness*):
The selection function *sel* is well-defined for each $elem1 \in c1.ae$
PO 2 (*attributes commute*):
$c1.m\_m = c2 \text{ and } c1.ae\text{->}includes(elem1) \text{ implies } elem.att = c2.ae\text{->}sel(elem1).att$
for all attributes *att* that are visible in *T*.
PO 3 (*outgoing associations commute*):
$c1.m\_m = c2 \text{ and } c1.ae\text{->}includes(elem1) \text{ implies } elem.ae1.m\_m = c2.ae\text{->}sel(elem1).ae1$
for all association ends *ae1* of associations that are outgoing from class *T* and that are not a composition.

4.3.2 Application of the algorithm

As an example, we analyze now the concrete syntax definition given in Sect. 3.1. As we shall see, the analysis process will be an enormous help for the language designer to discover losses in the modeling language metamodel.

Step 1 We have two display manager classes.

`ClassifierDM` The assignment is $cdm1.me.m\_m := cdm1.m\_r.me$
PO 1 (*attribute* `name` *commutes*):
$cdm1.m\_r = cdm2 \text{ implies } cdm1.me.name = cdm2.me.name$
This conjecture follows from invariant `label` and the fact that the composed function *vo.label.text* commutes for bijection *m_r*:
$cdm1.m\_r = cdm2 \text{ implies } cdm1.vo.label.text = cdm2.vo.label.text$
PO 1 (*attribute* `isAbstract` *commutes*):
$cdm1.m\_r = cdm2 \text{ implies } cdm1.me.isAbstract = cdm2.me.isAbstract$
The argumentation is the same as for attribute `name`.
PO 2 (*outgoing associations commute*): not applicable

`AssciationDM` The assignment is $adm.me.m\_m = adm.m\_r.me$
PO 1 (*attribute* `name` *commutes*):
$adm1.m\_r = adm2 \text{ implies } adm1.me.name = adm2.me.name$
This follows from invariant `roles` and the fact that *vo.label* commutes for *m_r*:
$adm1.m\_r = adm2 \text{ implies } adm1.vo.label = adm2.vo.label$
PO 2 (*outgoing associations commute*): not applicable

Step 2 We have to define for each object of modeling classes with outgoing compositions the mapping *m_m* on the children of this object. The ordering of this analysis does not matter for the current example, but can in some rare cases also be important for discarding proof obligations. Since the analysis for class `Association` is simpler than for `Classifier`, we start with `Association`.

`Association` We define selection function *sel* as
$as1.m\_m.ae\text{->}sel(elem1) := as1.m\_m.ae\text{->}at(as1.ae.indexOf(elem1))$
for each $elem1 \in as1.ae$ Informally speaking, if *elem1* is the first/second element in the sequence *as1.ae*, then *sel* yields also the first/second element of sequence *as1.m_m.ae*
PO 1 (*well-definedness*):
The selection function *sel* is well-defined simply because of multiplicity 2 of association end `ae`.
PO 2 (*attribute* `name` *commutes*):
$as1.m\_m = as2 \text{ and } as1.ae\text{->}includes(elem1) \text{ and } elem1.m\_m = elem2 \text{ and } elem2 = ae2.ae\text{->}sel(elem1) \text{ implies } elem1.name = elem2.name$

We need a case distinction whether *elem1* is the first or the second element in the sequence *as1.ae*. In both cases, the conjecture follows from invariant `roles`.

PO 3 (*outgoing association* `participant` *commutes*):

as1.m_m = as2 and as1.ae->includes(elem1) and elem1.m_m = elem2 and elem2 = ae2.ae->sel(elem1) implies elem1.participant.m_m = elem2.participant

We know that

as1.dm.vo.source.classifierDM.m_r = as2.dm.vo.source.classifierDM

The conjecture follows from this fact and the definition of m_m for Obj^1_C.

`Classifier` We define selection function *sel* as

c1.m_m.attribute->sel(elem1) := c1.m_m.attribute->any(a | a.name = elem1.name)

for each *elem1* ∈ *c1.attribute*

PO 1 (*well-definedness*):

This proof obligation is not provable what witness an error in the original concrete syntax definition. In order to resolve the problem, the language designer could add the following well-formedness rule to the metamodel of the modeling language, saying, that the name of an `Attribute` is unique within a `Classifier`:

```
context Classifier inv uniqueAttributeName:
self.attribute->unique(a| a.name)
```

PO 2 (*attribute* `name` *commutes*):

c1.m_m = c2 and c1.attribute->includes(elem1) implies elem1.name = c2.attribute->sel(elem1).name

Simply by construction of *sel*.

PO 3 (*outgoing association* `type` *commutes*):

c1.m_m = c2 and c1.attribute->includes(elem1) implies elem1.type.m_m = c2.attribute->sel(elem1).type

We know from invariant `attributes` that

elem1.name.concat(' :').concat(elem1.type.name) = c2.attribute->sel(elem1).name.concat(' :').concat(c2.attribute->sel(elem1).type.name)

Under the assumption that names of `Attribute` and `Classifier` do not contain the character '`:`' we could conclude that

elem1.type.name = c2.attribute->sel(elem1).type.name

However, this is not sufficient to justify the conjecture. The metamodel still misses some well-formedness rules. In order to make the conjecture provable, the language designer could add the following well-formedness rules to the metamodel:

```
context Classifier inv delimiterClassifierName:
    Set{1..self.name->size()-1}->forAll(i|
      self.name.substring(i,i+1) <> ':')
context Attribute inv delimiterAttributeName:
    Set{1..self.name->size()-1}->forAll(i|
      self.name.substring(i,i+1) <> ':')
context Classifier inv uniqueClassifierName:
    Classifier.allInstances()->unique(c| c.name)
```

4.3.3 Correctness of analysis algorithm

The algorithm defines a mapping *m_m* on the objects of the modeling part of *csi1* to objects of the modeling part of *csi2*. This mapping is, first of all, a total function and covers all objects of *csi1*, because the algorithm starts assigning *m_m* for all root objects (which are connected with a display manager object) and traverses for each class the composition links. Since we assume that each non-root object in *csi1* has an owner (a container in the composition hierarchy) the algorithm finally reaches each object in *csi1* and defines *m_m* for it.

The mapping *m_m* is injective and surjective because of the generated proof obligation *PO 1* (*well-definedness*).

Finally, also the fact that all attributes and outgoing links commute over *m_m* is ensured by the generated proof obligations.

4.4 Encoding of proof obligations for SIMPLIFY

Solving proof-obligations manually is an error-prone and tedious task. On the other hand, there are currently no automatic deduction systems available that could solve the above proof obligations formulated in OCL.

In the sequel, we show by example how our proof obligations can be encoded into first-order logic so that automated deduction tools (we have used the decision procedure SIMPLIFY [10]) can prove or disprove the generated proof obligation. SIMPLIFY was originally developed to decide the validity of a given formula in the theory of *Presburger Arithmetik* [11], 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 have been generated as the encoding of our correctness criterion, however, SIMPLIFY was impressively powerful and could prove or disprove almost every proof obligation. A very useful feature of SIMPLIFY is to return a counterexample when the proof goal has been disproved. This happens when the concrete syntax definition is not correct and the generated proof obligations are not valid.

Listing 4 shows the full encoding of the correctness criterion for the example given in Fig. 8. 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 in Listing 4, is the input file for the KeY system [12], which can be used as a front-end for SIMPLIFY since the KeY system is able to automatically generate equivalent input files for SIMPLIFY.

Listing 4 Encoding of correctness criterion for SIMPLIFY in KeY format

```
\sorts {
   classifier;
   classifierdm;
   labeledtextcontainer;
   textfield;
   string;
}
\functions{
  // associations
  classifier me(classifierdm);
  classifierdm dm(classifier);
  labeledtextcontainer vo(classifierdm);
  textfield label(labeledtextcontainer);
  // reverse mappings for vo, label
  classifierdm vorev(labeledtextcontainer);
  labeledtextcontainer labelrev(textfield);
  // attributes
  string name(classifier);
  string text(textfield);
}
\predicates{
  // attributes
  isAbstract(classifier);
  inItalic(textfield);
  // encoding of bijection m_r
  mapClassifier(classifier,classifier);
  mapClassifierDM(classifierdm,classifierdm);
  mapLabeledTextContainer(labeledtextcontainer,
                          labeledtextcontainer);
  mapTextField(textfield,textfield);
}
\problem {
// premise
// invariant on ClassifierDM (core of syntax definition)
(\forall classifierdm cdm; name(me(cdm)) =
   text(label(vo(cdm)))) &
// properties of mapClassifierDM
(\forall classifierdm cdm1; \forall classifierdm cdm2;
   (mapClassifierDM(cdm1,cdm2)
     -> mapLabeledTextContainer(vo(cdm1), vo(cdm2)))) &
// properties of mapLabeledTextContainer
(\forall labeledtextcontainer ltc1;
   \forall labeledtextcontainer ltc2;
     (mapLabeledTextContainer(ltc1,ltc2)
       -> mapClassifierDM(vorev(ltc1), vorev(ltc2)) &
          mapTextField(label(ltc1), label(ltc2)))) &
// properties of mapTextField
(\forall textfield tf1; \forall textfield tf2;
   (mapTextField(tf1,tf2)
     -> mapLabeledTextContainer(labelrev(tf1),
                                labelrev(tf2)) &
        text(tf1) = text(tf2) &
        (inItalic(tf1) <-> inItalic(tf2))))
->
// conclusio
// PO to show that attributes name, isAbstract commute
\forall classifierdm cdm1;\forall classifierdm cdm2;
  (mapClassifierDM(cdm1,cdm2)
    -> name(me(cdm1)) = name(me(cdm2)) &
       (isAbstract(me(cdm1)) <-> isAbstract(me(cdm2))))
}
```

The KeY syntax expects at the beginning of the file a list of declarations for all occurring types, functions and predicates. There are standard techniques how a UML class diagram is encoded by types, functions and predicates, for example, the classes are represented by types, associations by functions and attributes by functions or predicates (see [12] for details). The clause `\problem` contains the formula to be proven 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 problem clause is a conjunction of all the invariants occurring in the concrete syntax metamodel and the commute-property of all attributes and association ends with respect to bijection *m_r* (which is encoded here as a family of map-predicates). The conclusio encodes literally proof obligation *PO 1* in *Step 1* (see Sect. 4.3.1) for attributes `name` and `isAbstract`. The encoding of all remaining proof obligations is analogous.

When invoked for this input file shown in Listing 4, SIMPLIFY cannot find a proof because the original concrete syntax definition 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. 8. Such counterexamples are extremely valuable for the language designer to detect and to resolve errors in the concrete syntax definition.

4.5 Summary

In this section, we presented two techniques for analyzing the correctness of concrete syntax definitions. Technique 1 simply extends explicitly *m_r* to *m_m* on the modeling part of *csi1*. It has to been shown manually that the defined mapping *m_m* is actually a bijection and that all attribute values and outgoing association links commute.

Technique 2 defines the mapping *m_m* only implicitly. The algorithm implementing Technique 2 starts with the root objects in *csi1* (those, that are connected with a display manager object) and traverses then the tree structure of the modeling part (exploiting our additional assumption that each object in the modeling part has a container object or is a root object). Whenever *m_m* is extended to the children objects of a given object, some proof obligations are generated ensuring that the defined mapping *m_m* is indeed a bijection and commutes the values for relevant attributes and outgoing links.

We have seen on the example, how the analysis process can help the language designer to uncover forgotten

well-formedness rules in the modeling part, i.e. errors in the definition of the modeling language. Many of these errors are found automatically by modern deduction tools, such as SIMPLIFY.

5 Related work

This paper has two topics, the formalization and the analysis of concrete syntax definitions. While the definition of the concrete syntax for modeling languages has attracted recently much research interest, the correctness analysis of such definitions is a rather new topic.

The definition of textual/graphical representation languages has a long tradition in computer science. Textual languages are most often defined by an EBNF grammar, whereas graphical languages are either defined by graph grammars [13] or directly in form of a metamodel (see, for example, the OMG standard *Diagram Interchange* [7] or the language definitions given by Costagliola et al. in [5]). A technique to bridge graph grammars and metamodeling is described by Bardohl et al. in [14]. There are even various generators for graphical editors available (e.g., GenGed [15], AToM^3 [16], DiaGen [17], to name few of them), but the language specification used by these generators usually do not have such a strict separation between modeling language and representation language as we described in this paper. A notably exception is DiaGen, which provides a pre-defined representation language consisting of boxes, lines, etc. Diagrams in this representation language are internally represented in form of a *hypergraph model* (HGM), what would correspond in our setting to an instance of the representation language metamodel. The bridge between an HGM and an instance of the modeling language metamodel is described in DiaGen by *reduction rules*, which allow a stepwise construction of the model from an HGM. Recall that in our approach this bridge is specified by OCL constraints, which are usually attached to display manager classes.

Another approach for bridging the gap between modeling and representation language is to use a template-based language, e.g., the OMG standard *MOF Models to Text Transformation Language* [18], in order to describe a mapping from the model to the representation of the model. Also general purpose model transformation languages, such as MTL, ATL, or QVT could be used to describe such mappings. While mapping approaches work sufficiently well in practice, we are not aware of any research that investigates the correctness of such mapping definitions. Note that mappings, which generate the model representation out of the model, are generally in danger to assign two different models to the same representation. In this case, the representation becomes ambiguous.

Xia and Glinz present in [19] an approach to describe the concrete syntax of their own graphical modeling language ADORA [20]. The main idea is to map the graphical representation of a language construct to a textual representation and to define the textual 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.

Alanen et al. define in [21] the special purpose transformation language *DIML*, which can render models written in an arbitrary modeling language into the representation language *Diagram Interchange* [7]. However, DIML-transformations are strictly less expressive than our concrete syntax definitions since they map each model to a unique representations and sacrifice—by doing so—presentation options. Moreover, there is a simple technique to encode DIML-transformations into our concrete syntax definitions (for each DIML-rule, a new display manager class had to be created). Consequently, our algorithm to analyze the correctness of syntax definitions could also be applied to show the correctness of DIML-transformations.

Muller et al. propose in [9] an interesting approach to incorporate the mapping from a textual representation of a model to the model itself already into the definition of the textual language, i.e. into the corresponding EBNF grammar. Their main technique is to define a metamodel for EBNF grammar rules. Besides metaclasses representing traditional EBNF constructs, this metamodel has also a metaclass called *Template*, whose purpose is to realizes the bridge from the textual representation elements to the concepts of the modeling language.

Last but not least, we have to mention Triple-Graph-Grammars (TGGs), invented by Schürr already in 1994 [22] (see also [23] for a more recent survey and a case study). Our approach to define a concrete syntax shares many similarities with the TGG approach. The most important difference lies in the fact, that our goal is merely to describe valid instances of the concrete syntax metamodel, but we are—in the first place—not interested in how such instances are constructed. This is the goal of TGGs and causes some restrictions for the kind of constraints one can attach to bridging elements (only equations are allowed). Nevertheless, our correctness analysis algorithm described in Sect. 4 could also be applied to verify, that isomorphic structures on the right-hand-side of TGG instances are always mapped to isomorphic structures on the left-hand-side.

6 Summary

In this paper, we have described a purely declarative approach to formally define the concrete syntax of modeling languages. The formalization we propose is directly based on the primary language definition, i.e. the metamodel that encodes the abstract syntax, and on a metamodel of the representation language. While the user is free to choose any convenient

metamodel of the representation language he has in mind, there are already a number of frameworks for metamodels of visual languages available, e.g. the OMG standard for Diagram Interchange [7] or the Graphical Model Framework (GMF) from Eclipse [24].

A formal definition of a concrete syntax has many advantages over an informal one. First of all, it forces the language designer to strictly distinguish between the concepts a modeling language should provide and the representation of these concepts. Furthermore, we have shown how a formal definition can be rigorously analyzed, what uncovers problems either of the concrete or of the abstract syntax (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.

Acknowledgements I would like to thank the anonymous reviewers for their insightful comments and for guidance on earlier drafts of this paper.

References

1. Marković, S., Baar, T.: An OCL semantics specified with QVT. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) Proceedings, MoDELS/UML 2006, Genova, Italy, October 1–6, 2006, vol. 4199, LNCS, pp. 660–674. Springer, Heidelberg (2006)
2. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Object Technology Series, 2nd edn. Addison-Wesley, Reading (2005)
3. Baar, T.: Correctly defined concrete syntax for visual models. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) Proceedings, MoDELS/UML 2006, Genova, Italy, vol. 4199, LNCS, pp. 111–125. Springer, Heidelberg (2006)
4. Fondement, F., Baar, T.: Making metamodels aware of concrete syntax. In: Hartman, A., Kreische, D. (eds.) Proceedings of European conference on Model Driven Architecture (ECMDA-FA), vol. 3748, LNCS, pp. 190–204. Springer, Heidelberg (2005)
5. Costagliola, G., De Lucia, A., Orefice, S., Polese, G.: A classification framework to support the design of visual languages. J. Vis. Lang. Comput. **13**(6), 573–600 (2002)
6. Costagliola, G., Deufemia, V., Polese, G.: A framework for modeling and implementing visual notations with applications to software engineering. ACM Trans. Softw. Eng. Methodol. (TOSEM) **13**(4), 431–487 (2004)
7. OMG. Diagram Interchange, Version 1.0. formal/06-04-04 (2006)
8. USE homepage, http://www.db.informatik.uni-bremen.de/projects/USE/
9. Muller, P.-A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J.-M.: Model-driven analysis and synthesis of concrete syntax. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio G. (eds.) Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1–6, 2006, Proceedings, vol. 4199, LNCS, pp. 98–110. Springer, Heidelberg (2006)
10. Detlefs, D.L., Nelson, G., Saxe, J.B.: Simplify: the ESC theorem prover. Technical Report (1996)
11. Presburger, M.: Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchen, die addition als einzige operation hervortritt. Sprawozdanie z I Kongresu Matematikow Krajow Slowcanskich Warszawa, pp. 92–101 (1929)
12. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach, vol. 4334, LNCS. Springer, Heidelberg (2007)
13. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1. World Scientific, Singapore (1997)
14. Bardohl, R., Ehrig, H., de Lara, J., Taentzer, G.: Integrating metamodelling aspects with graph transformation for efficient visual language definition and model manipulation. In: Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, vol. 2984, Lecture Notes in Computer Science, pp. 214–228. Springer, Heidelberg (2004)
15. Bardohl, R.: A visual environment for visual languages. Sci. Comput. Program. (SCP) **44**(2), 181–203 (2002)
16. de Lara, J., Vangheluwe, H.: Atom^3: A tool for multi-formalism and meta-modelling. In: Kutsche, R.-D., Weber, H. (eds.) Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8–12, 2002, Proceedings, vol. 2306, LNCS, pp. 174–188. Springer, Heidelberg (2002)
17. Minas, M.: Specifying graph-like diagrams with DiaGen. Electron. Notes Theor. Comput. Sci. **72**(2), 102–111 (2002)
18. OMG. MOF Models to Text Transformation Language, Final Adopted Specification. OMG Adopted Specification, ptc/06-11-01 (2006)
19. Xia, Y., Glinz, M.: Rigorous EBNF-based definition for a graphic modeling language. In: Proceedings of 10th Asia-Pacific Software Engineering Conference (APSEC 2003), pp. 186–196. IEEE Computer Society Press (2003)
20. Glinz, M., Berner, S., Joos, S.: Object-oriented modeling with ADORA. Inf. Syst. **27**(6), 425–444 (2002)
21. Alanen, M., Lundkvist, T., Porres, I.: A mapping language from models to DI diagrams. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) Proceedings, MoDELS/UML 2006, Genova, Italy, vol. 4199, LNCS, pp. 454–468. Springer, Heidelberg (2006)
22. Schürr, A.: Specification of graph translators with triple graph grammars. In: Proceedings of 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'94), vol. 903, LNCS, pp. 151–163. Springer, Heidelberg (1995)
23. Königs, A., Schürr, A.: Tool integration with triple graph grammars—a survey. In: Heckel, R. (ed.) Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques, vol. 148, Electronic Notes in Theoretical Computer Science, pp. 113–150. Elsevier, Amsterdam (2006)
24. Eclipse. Eclipse project, http://www.eclipse.org (2007)

Author Biography

Thomas Baar This paper was written while Thomas was employed by École Polytechnique Fédérale de Lausanne (EPFL), Switzerland. For researchers, EPFL is a place almost in heaven: enthusiastic students, moderate teaching load, a de-facto unlimited budget for attending conferences; all this combined with a high degree of scientific freedom. Moreover, the professional duties still allow to devote plenty of time to leisure activities, to the family, and to teach the next generation some core concepts of the real world, and how these concepts are represented.