Squeezer Graphs: Proposing a Simple Basis for Computing (Meta-)Model Features

Work-in-progress Paper

Martin Gogolla

Computer Science Department, University of Bremen, Bremen, Germany
gogolla@informatik.uni-bremen.de

Abstract. The paper introduces so-called squeezer graphs that are used to represent models and their instantiations. In a squeezer graph, multiple modeling levels are squeezed into a single graph. Properties of models that span multiple levels like typing uniqueness and other features can be computed in them in a direct way.

1 Introduction

In recent years, modeling and metamodeling of software has become a major topic for research and development, for example in the context of languages like UML (Unified Modeling Language) [9], which includes the OCL (Object Constraint Language) [10]. A model or a metamodel is said to abstract unnecessary details. When using a metamodel one typically has different, at least three levels of abstraction. A single model (or metamodel) is often formally described, but less attention is paid to formally describe the connection between a model and its instantiations or between a metamodel and its models. Frequently, it is said that there is an `instanceOf` relationship between the levels. Figure 1 shows an example for modeling with three levels. The challenge in this contribution is to introduce a formal approach that utilizes graphs and that covers all models from all levels and, at the same time, the relationship between the models.

This contribution basically has a very simple structure: it introduces one UML class model including restricting OCL constraints and two example object models. It also shows the implementation of the three artifacts within the UML and OCL tool USE[1]. The research contribution from this paper lies in the class model that represents a new, previously not studied, object-oriented model for graphs and its application to models with multiple levels. In particular it permits graph items that can be nodes or edges at the same time. Thus, a kind of a hybrid *G*raph it*em*, for short a *Gem*, is supported. The edges are hyperedges [6] with possibly many source and many target nodes. The defined graphs can be used for determining the interpretation of models ranging over multiple levels and are called squeezer graph, because they allow a developer to squeeze several conceptual modeling levels into one graph structure.

[1] https://sourceforge.net/projects/useocl/

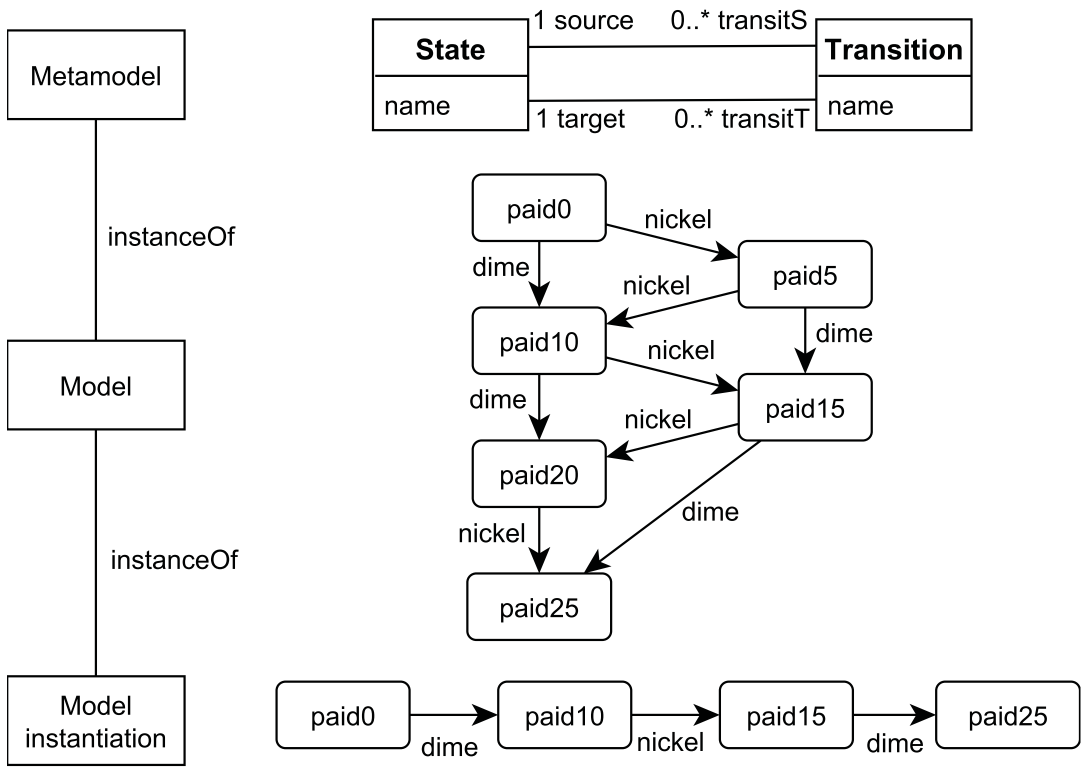


Fig. 1. Metamodel, model, model instantiation.

On the technical side, our approach takes advantage of two valuable abstraction features available in object-oriented modeling and in the tool that we employ: (a) inheritance on classes and associations utilizing such generalized classes, i.e., a `NodeEdge` object can be used as a `Node` and as an `Edge` dependent on the needed context; (b) derived associations that allow to abstract away unneeded connecting objects, i.e., a typing `Edge` object together with a `Source` link and a `Target` link can be represented as a derived link graphically represented in an eye-catching way.

The rest of this contribution is structured as follows. Section 2 studies a first, basic example for modeling in which both node-like and edge-like information is typed, and it presents the corresponding squeezer graph. Section 3 focuses on the general graph model behind squeezer graphs. Section 4 treats a second example with four levels of modeling and its squeezer graph representation. Section 5 shows how computations in squeezer graphs can be realized by utilizing OCL expressions. Section 6 ends the paper with concluding remarks and future work.

2 Simple Squeezer Graph: Gem Used as Node and Edge

Figure 2 shows an introductory example in form of a conceptual sketch in Fig. 2(a) that explains the basic idea behind squeezer graphs. In the upper row of Fig. 2(a), one can identify a class model structure and in the lower row a corresponding object model structure, like in [9]. All elements from the lower part are typed through elements from the upper part. Taking a view from graphs, one could say that instance-like nodes are typed by type-like nodes, and an edge

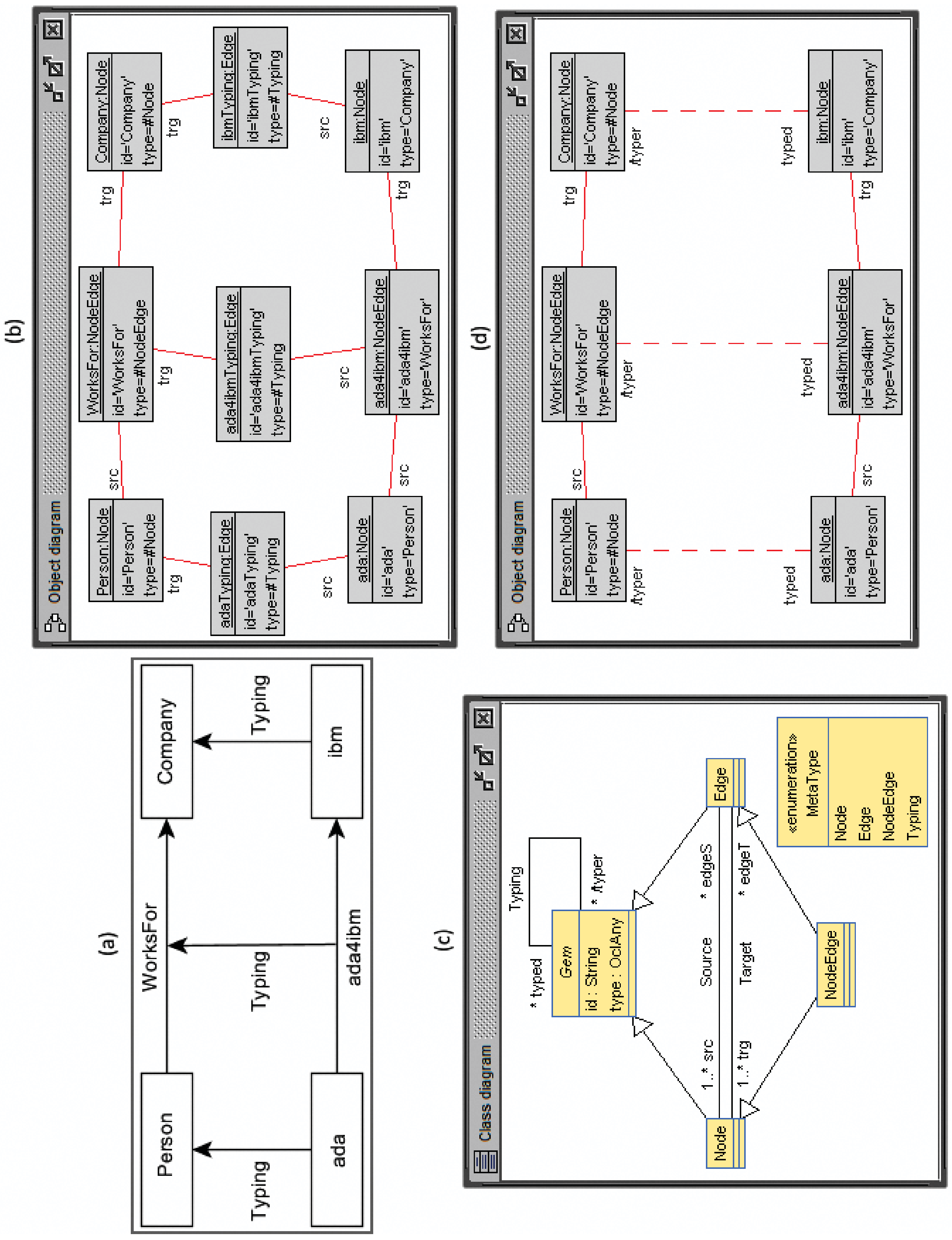


Fig. 2. Two-level conceptual graph, squeezer graph and squeezer graph class model.

of the instance level is typed by an edge of the type level. The task of a squeezer graph is to formally represent the complete information from the conceptual, informal sketch in terms of a single graph. Squeezer graphs squeeze type levels and instance levels into one single graph.

Figure 2(b) shows the formal representation of the conceptual sketch in form of an object model that belongs to the class model in Fig. 2(c). All information is presented in form of `Node`, `Edge`, and `NodeEdge` objects: the lower instance level, the upper type level and the connecting typing level. The fact that an instance level graph area is typed by a type level graph area becomes apparent by the role names on the links that indicate a source and target direction. A point of special attention is the fact that the object `WorksFor` plays both the role of a `Node` and the role of an `Edge`. The same is true for the object `ada4ibm`. Therefore both objects must belong to the class `NodeEdge`. Objects belonging to class `Gem` and its specializations (where `Gem` is short for *G*raph it*em*) constitute the elements in a squeezer graphs.

The class model will be discussed in detail in the next section. The class model formally introduces (a) the used classes and associations, (b) the single enumeration and (c) the defined inheritance structure for squeezer graphs.

Definition: A **squeezer graph** is defined as an object model for the class model in Fig. 2(c); the object model has to satisfy the additional invariants to be defined in Sect. 3.

In contrast to many works in the graph transformation and graph computation area, that typically define a particular notion of graph in a mathematical style as a tuple with k components satisfying particular requirements (e.g., $G = (N, E)$), we use a style for the definition of a graph that is influenced by the modeling style developed in recent years within software engineering.

Figure 2(d) displays a variation of the object model where the typing part with the `Edge` objects is hidden in favor of representing the typing information directly as derived links in dashed form between the typing and the typed `Node` objects with roles `typer` and `typed`. In the literature on modeling with multiple levels [8, 2, 7], very often a visual representation that is close to this style is preferred. This representation is also closer to the conceptual sketch.

3 Squeezer Graph Class Model with OCL Constraints

Let us go through the class model in Fig. 2 in a systematic way. The class model involves three concrete classes `Node`, `Edge`, and `NodeEdge` and the abstract class `Gem` (*G*raph it*em*). An `Edge` object is a hyperedge with possibly many `Node` objects as sources and possibly many `Node` objects as targets, as expressed by the two associations `Source` and `Target`. Inheritance is defined as displayed. Thus `NodeEdge` objects can be used at the same time as `Node` objects and as `Edge` objects. All `Gem` objects possess an attribute `id` and an attribute `type`. The `id` is a `String`, and the `type` is an `OclAny` [10] that will be restricted however through a constraint to come from the enumeration `MetaType` or to be a `String`.

In Fig. 3 we show the general schema for a squeezer graph with three levels where each level is enclosed by a light gray rounded box. The top-most level 1 has only `Gem` objects possessing `type` values `#Node`, `#Edge`, or `#NodeEdge`. In the level 2 below the top-most level, the `Gem` objects will have `type` values which are equal to the `id` values from level 1. This typing is represented by `Edge` objects that are directed from level 2 to level 1. On level 3 the `Gem` objects will take `type` values from upper levels; as pictured here from level 2, but the `Edge` objects could point also from level 3 to level 1.

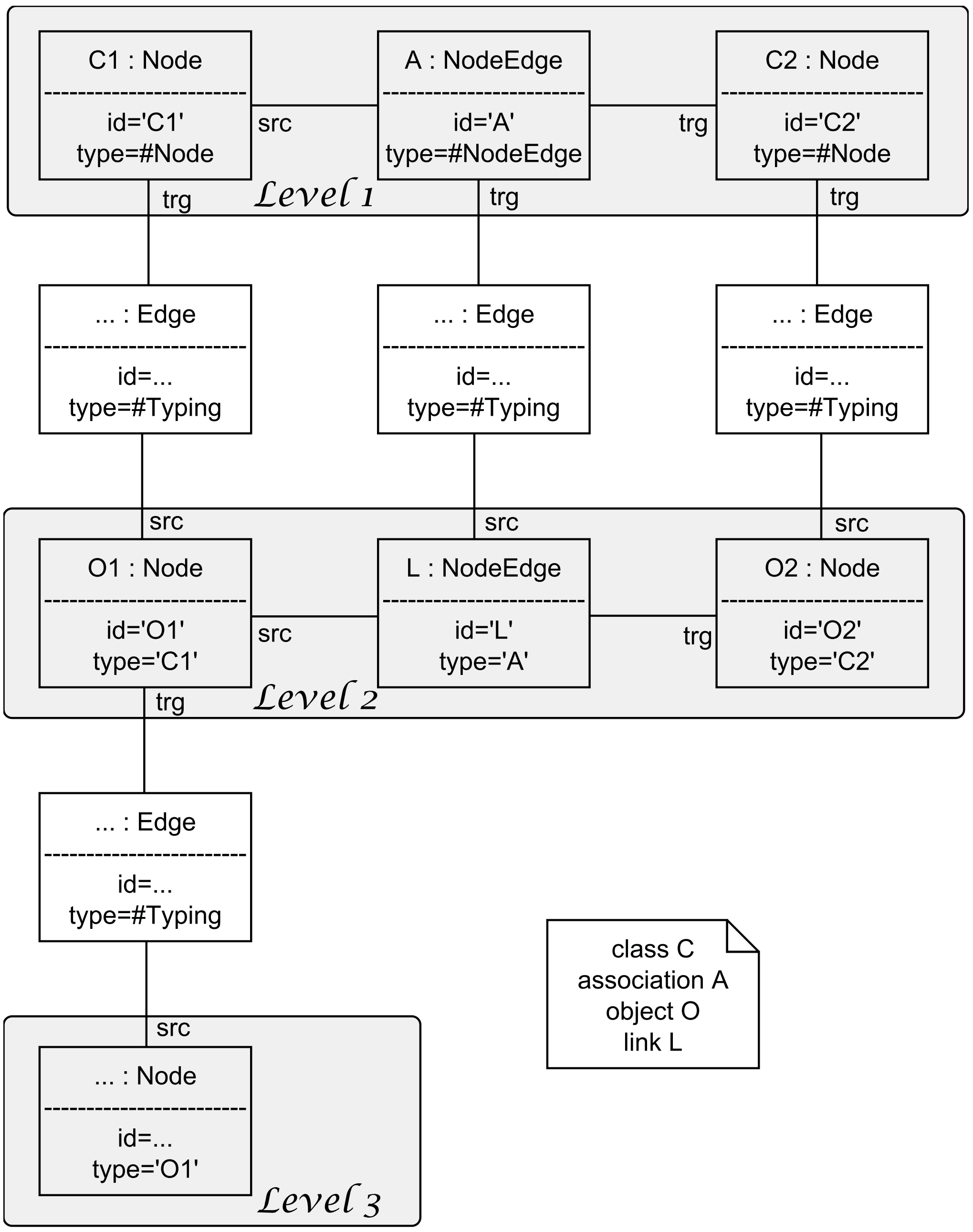


Fig. 3. General schema for a squeezer graph with three levels.

For the time being, there are the following invariants restricting the squeezer graph class model. More invariants, e.g., for stronger typing restrictions, could be added.

– The invariant `uniqueId` guarantees that the `id` value is unique over all `Gem` objects.

```
context g1,g2:Gem inv uniqueId:
  g1<>g2 implies g1.id<>g2.id
```

– The expression `type_String_MetaType` restricts the `type` attribute to `String` or the enumeration `MetaType`.

```
context g:Gem inv type_String_MetaType:
  g.type.oclIsTypeOf(String) or g.type.oclIsTypeOf(MetaType)
```

- The constraint typeString_IMP_uniqueGemExists requires that if the type is a String, then a unique Gem with that type value exists.

  ```
  context g:Gem inv typeString_IMP_uniqueGemExists:
    g.type.oclIsTypeOf(String) implies
      Gem.allInstances->one(h | h.id=g.type)
  ```

- The condition typeTyping_IMP_edge demands that if the type value is equal to the literal #Typing from the enumeration MetaType, then the Gem object is an Edge object.

  ```
  context g:Gem inv typeTyping_IMP_edge:
    g.type=#Typing implies g.oclIsTypeOf(Edge)
  ```

- The invariant usedAsNodeAndEdge restricts the use of NodeEdge objects as it requires that such objects must be used in the associations as Node and as Edge objects.

  ```
  context ne:NodeEdge inv usedAsNodeAndEdge:
    (ne.edgeS->notEmpty or ne.edgeT->notEmpty) and
    ne.src->notEmpty and ne.trg->notEmpty
  ```

The derived association Typing allows to hide Edge objects that possess the type value #Typing. Instead of displaying an Edge object with two links, it becomes possible to display the information about #Typing through a single dashed, derived link that indicates the typer and the typed.

```
association Typing between
  Gem [0..*] role typer derived =
    self.oclAsType(Node).edgeS->select(type=#Typing).trg->asSet()
  Gem [0..*] role typed
end
```

Two constraints refer to the derived association Typing:

- The constraint type_Node_Edge_NodeEdge_IMP_hasNoTyper refers to the derived association and requires that the Node, Edge and NodeEdge objects from the top-most level are not participating in the derived association Typing in the role typed.

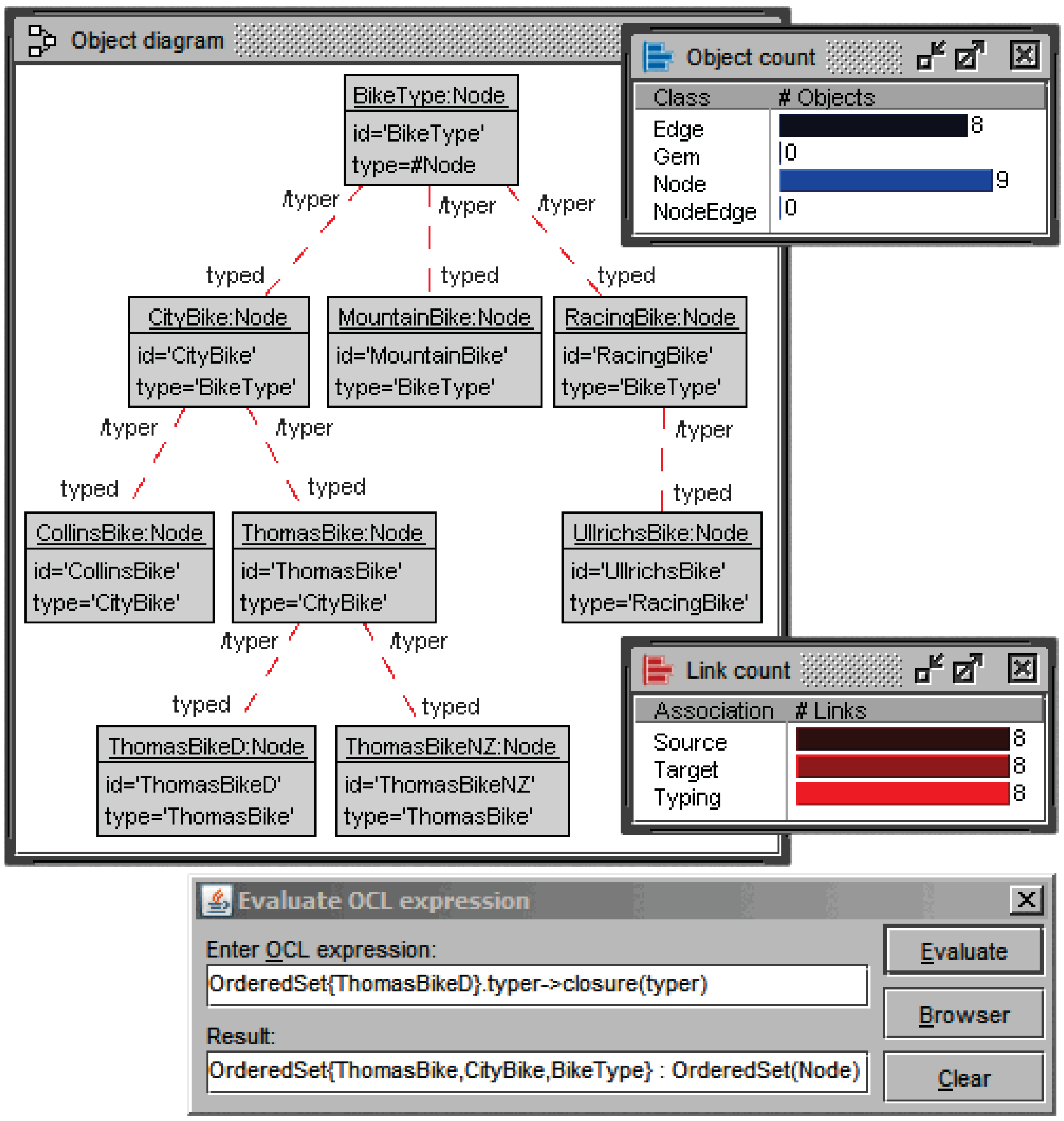
  ```
  context g:Gem inv type_Node_Edge_NodeEdge_IMP_hasNoTyper:
    (g.type=#Node or g.type=#Edge or g.type=#NodeEdge) implies
      g.typer->isEmpty
  ```

- The invariant type_String_IMP_hasTyper demands that if the type of a Gem object is a String, the Gem object must have a typer in the association Typing.

  ```
  context g:Gem inv type_String_IMP_hasTyper:
    g.type.oclIsTypeOf(String) implies g.typer->notEmpty
  ```

4 Four-level Squeezer Graph

Figure 4 introduces a squeezer graph with four modeling levels. This example about bicycles and bicycle types is inspired by the multi-level modeling workshop challenge [3] in 2017. Figure 4 shows `Node` objects, whereas `Edge` objects or links for the associations `Source` or `Target` are not displayed in the figure, but they are present in the object model: the typing information is given through derived `Typing` links with roles `typer` and `typed`.



Class	# Objects
Edge	8
Gem	0
Node	9
NodeEdge	0

Association	# Links
Source	8
Target	8
Typing	8

Enter OCL expression:

```
OrderedSet{ThomasBikeD}.typer->closure(typer)
```

Result:

```
OrderedSet{ThomasBike,CityBike,BikeType} : OrderedSet(Node)
```

Fig. 4. Squeezer graph over four levels for bicycles.

On the top-most level 1, a general `BikeType` is defined as a `Node` object. On level 2 more specialized bicycles representing `CityBike`, `MountainBike`, and `RacingBike` vehicles are provided. In order to understand level 3, first imagine that instances of the previously mentioned types are defined: `CollinsBike` and `ThomasBike` as instances of `CityBike` and `UllrichsBike` as an instance of `RacingBike`. However, `ThomasBike` is not merely an instance, but also a type having on the next lower level two instances `ThomasBikeD` and

ThomasBikeNZ (because Thomas lives in Germany and New Zealand and therefore has two bikes). In this squeezer graph, instances and types co-exist on the same level.

In our tool USE, the graph behind the object diagram can be inspected and analyzed further. Figure 4 shows information on the objects and links. The Object count window tell us, that apart from Node objects also Edge objects are present, but currently not visible. The Link count windows explains that each Edge object must be participating in the Source and Target association. Last but not least, the Evaluate OCL expression window computes by means of the closure operation from OCL for the Node object ThomasBikeD all its types and the type of its types in the correct order by using an ordered set: ThomasBike, CityBike, BikeType.

5 Computing Graph Features and Properties

We have defined in the previous sections a graph model that allows the developer to represent different modeling levels and their connections in a unifying way as a squeezer graph. Since a squeezer graph is an object model belonging to an ordinary UML and OCL class model, we can employ OCL to derive and compute properties and features of the object model. The formulation of invariants for the graph model with OCL can already be understood as a means for computing features and properties of the defined graphs.

We have already shown in the previous section how to compute for a concrete object the several types to which the object belongs: the OCL expression `OrderedSet{ThomasBikeD}.typer->closure(typer)` in Fig. 4.

The examples we have presented up to now all returned a unique type in the next higher modeling level. This must not necessarily be the case, however. For example, one Node object may be engaged in two #Typing edges in the target direction. In order to evaluate whether or not the typing is unique, one can check the following OCL expression on the object in question: `obj.typer->size=1` will return a truth value and checks for a singleton set of types.

The examples shown so far also had the property that the #Typing edges were acyclic between the modeling levels. However, this must also not necessarily be the case. But one can check this property, again with an OCL expression: `Set{obj}.typer->closure(typer)->excludes(obj)` tests whether for the object obj the typing is acyclic.

Applied graph computation model: In summary, we propose to use UML as a model to define graph notions and the Object Constraint Language (OCL) as a model for the definition of graph computations and graph properties. We regard OCL as a suitable candidate for such a task, because it (a) is computationally complete (due to the option to define recursive operations), (b) offers various collections kinds that directly support concepts crucial within the graph area (e.g., a path in a graph can be represented by an `OrderedSet(Node)` or a `Sequence(Node)`), and (c) supports collections operations that are highly relevant in graph computations (e.g., the operation `closure` for computing the transitive closure of a given set of edges).

In this contribution, we have been focusing on a particular notion of graph, namely our squeezer graphs. However, multiple other notions of graphs could be defined in the same style with UML and OCL (we only give two examples):

– If one does not need or want the notion of a hyperedge and one wants to work only with “plain” edges, then one can restrict our graph model by simply requiring:
  `Edge.allInstances->forAll(e | e.src->size=1 and e.trg->size=1).`
– If one is not interested in different graph levels or modeling levels at all and one wants to work with “plain” graphs (sketched as $G = (N, E)$ with $E \subseteq (N \times N)$), then one can achieve this with the following restrictions (apart from excluding hyperedges as done above):
  (a) `NodeEdge.allInstances->isEmpty()` and
  (b) `Gem.allInstances->forAll(g | g.type=#Node or g.type=#Edge).`

6 Related Work

We only mention few approaches. One aim with this workshop paper is to obtain feedback from the graph community about similar approaches. In previous own work [5] we have discussed a simple version of squeezer graphs however without the use of inheritance which definitively enriches the application possibilities. In [4] a related graph model has been proposed that however strictly distinguishes between nodes and edges. In [1] a conceptual model for connections between modeling levels is proposed, but a formalization in terms of a general graph-based approach is left open.

7 Conclusion

The problem discussed in this contribution has been how to formulate models and their connections with a graph in a direct way. We have defined a graph model in which a graph item can play both the role of a node and the role of an edge. We have demonstrated with several examples the applicability of the defined graphs.

Future work contains a number of topics. We want to compare our graph model with other proposals from the literature for flexible graphs with interesting features that could be used for models over multiple levels. We want to define these graph models with class models and obtain the graphs then as object models. One further direction that we see in our current proposal in order to model other graph aspects is an extension of the `MetaType` for capturing inheritance and containment: just as special edges are introduced for `#Typing` purposes, one could introduces special edges for inheritance relationships; and one could allow special containment relationships that express that a collection of graph items is contained in another node. Last but not least, larger case studies must check the applicability of the proposed approach.

References

1. Atkinson, C., Gerbig, R., Kühne, T.: A Unifying Approach to Connections for Multi-Level Modeling. In Lethbridge, T., Cabot, J., Egyed, A., eds.: 18th ACM/IEEE MoDELS, IEEE Computer Society (2015) 216–225
2. Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. IEEE Software **20**(5) (2003) 36–41
3. Clark, T., Frank, U., Wimmer, M.: Preface 4th Int. Workshop Multi-Level Modelling (MULTI 2017). In: Proc. MODELS 2017 Satellite Events, CEUR WS, Vol-2019. (2017) 210–212
4. de Lara, J., Guerra, E., Cobos, R., Moreno-Llorena, J.: Extending Deep Meta-Modelling for Practical Model-Driven Engineering. Comput. J. **57**(1) (2014) 36–58
5. Gogolla, M., Favre, J.M., Büttner, F.: On Squeezing M0, M1, M2, and M3 into a Single Object Diagram. In Baar, T., et al., eds.: Proc. MODELS Workshop on OCL, EPFL (Switzerland), LGL-REPORT-2005-001 (2005)
6. Habel, A.: Hyperedge Replacement: Grammars and Languages. LNCS 643. Springer (1992)
7. López-Fernández, J.J., Cuadrado, J.S., Guerra, E., de Lara, J.: Example-Driven Meta-Model Development. Software and System Modeling **14**(4) (2015) 1323–1347
8. OMG, ed.: Meta Object Facility, Version 2.5.1. OMG (2016) OMG Document, `www.omg.org`.
9. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language 2.0 Reference Manual. Addison-Wesley, Reading (2003)
10. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley (2003) 2nd Edition.