On Integrating
OCL and Triple Graph Grammars

Duc-Hanh Dang and Martin Gogolla

Department of Computer Science, University of Bremen
D-28334 Bremen, Germany
{hanhdd|gogolla}@informatik.uni-bremen.de

Abstract. Triple Graph Grammars (TGGs) tend to be a promising approach for explaining relationships between models in general, and model co-evolution and model consistency within model-driven development in particular. Declarative TGG descriptions can be translated into operational scenarios for model integration, model synchronization, and model transformation. To realize such scenarios, restrictions formulated with the Object Constraint Language (OCL) are an important factor. How to integrate TGGs and OCL is a topic of ongoing research activities. There are strong similarities between the result of such an integration and the Queries, Views and Transformations (QVT) standard of the Object Management Group (OMG). We propose a language for this integration: One part of this language has a one-one mapping to TGGs and the remaining part covers OCL concepts. The language is realized in our tool UML-based Specification Environment (USE) by taking two views on operations derived from TGG rules: Declarative OCL pre- and postconditions are employed as operation contracts, and imperative command sequences are taken as an operational realization. USE offers full OCL support for restricting models and metamodels with invariants, for checking pre- and postconditions of operations as well as for validating and animating transformation operations. Our approach covers a complete realization of TGGs incorporating OCL within our tool USE.

1 Introduction

Triple graph grammars (TGGs) have been first formally proposed in [1]. Their aim was to ease the description of complex transformations within software engineering. The original TGGs extend earlier ideas from [2]. With the advent of model-centric software development approaches (like UML) and model-driven engineering (MDE), TGGs tend to become one formal basis for underlying development concepts. TGGs and the recent QVT (Queries, Views, Transformations) share many functions and building blocks. But in contrast to TGGs, QVT includes the declarative language OCL (Object Constraint Language) which allows to express properties and to navigate in complex models.

In the recent literature, a need for an integration of TGGs and OCL has been expressed. However, such an integration has not been systematically studied or realized yet. Recent papers [3–7] often either underestimate the importance of OCL for constraints as pre- and postconditions of TGG rules or concentrate on non-deleting rules or ignore the automatic construction of metamodels for the correspondence domain of TGG rules. Within these works, OCL constraints are only employed for updating attribute expressions, whereas we also support OCL pre- and postconditions in TGG rules and OCL (meta-)-model invariants.

We propose a language for the integration of TGGs and OCL: One part of this language has a one-one mapping to TGGs and the remaining part covers OCL concepts. The language is realized in our tool UML-based Specification Environment (USE) [8] by taking two views on operational scenarios derived from a TGG rule: OCL pre- and postconditions are employed as operation contracts, and command sequences are taken as an operational realization. Our approach covers a complete realization of TGGs incorporating OCL within our tool USE as well as an on-the-fly verification of the operations.

The rest of this paper is structured as follows. Section 2 explains the basic idea by means of an example. Section 3 describes our realization in USE for the integration. Section 4 explains how the integration of TGGs and OCL can support model co-evolution and consistency. Section 5 comments on related work. The paper is closed with a summary and a conclusion.

2 The Basic Idea for Integrating TGGs and OCL

This section explains the basic idea of our approach by means of an example. We consider a typical situation within software development, where one wants to formally trace the relationship between a conceptual model and a design model. Figure 1 indicates the correspondence between an association end on the left side within a conceptual model and an attribute and an operation on the right side within a design model. For example, we recognize that the association end *customer* is mapped to the attribute *customer* and the operation *getCustomer*() of the class *Order*.

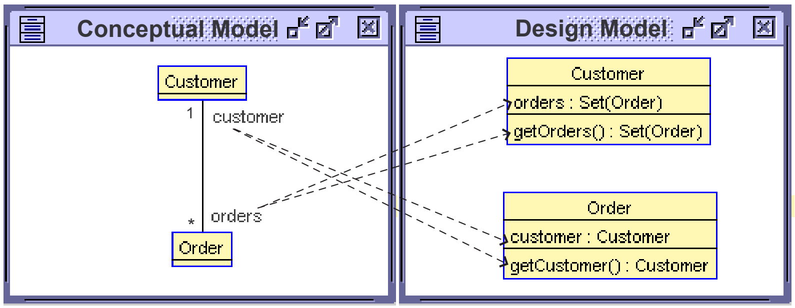

Fig. 1. Correspondence between Association Ends and Attributes

We want to express the requirement that, when we add an association between *Customer* and *Order* in the conceptual model, the corresponding classes in the design model are automatically extended with new attributes and operations. Therefore, a correspondence between the association ends in the conceptual model and the attributes and operations in the design model has to be established in a formal way.

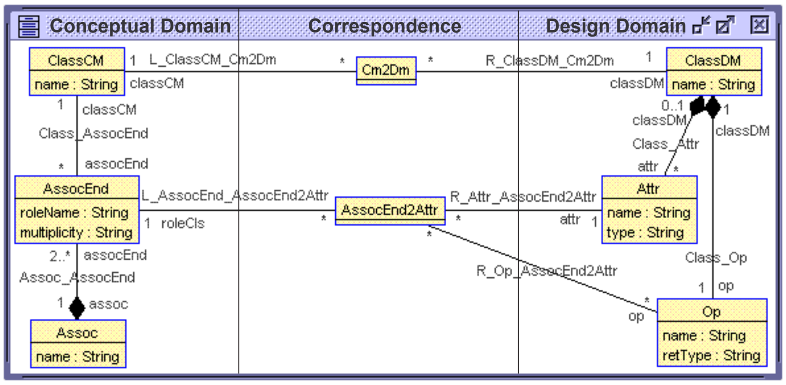

Fig. 2. Simplified Example Metamodels and their Connection

This scenario is a model transformation between UML class diagrams, and QVT [9] as well as TGGs can approach the requirement. Figure 2 shows the transformation in schematic form as a metamodel. The left side shows the metamodel for the conceptual domain, the right side displays the metamodel for the design domain, and the middle part pictures the connection resp. the correspondence between them. Later we will see, that the diagram in Fig. 1 can be seen as an instance diagram for the metamodels and that the dashed arrows (from Fig. 1) are represented as objects belonging to classes from the middle part. A TGG rule as well as QVT mappings can realize an action which adds an association and updates the attributes and operations as explained in the following.

2.1 QVT for the Example Transformation

Figure 3 presents the example transformation in the QVT Core language. The transformation contains two mappings, *cm2Dm* and *assocEnd2Attr*. The mappings declare constraints that must hold between the model elements within the transformation. The transformation requires two new types of elements (two new classes) for the correspondence domain: *Cm2Dm* and *AssocEnd2Attr*. The mappings can be executed by matching patterns which are declared within the '`check`' parts, i.e., domain patterns, and the '`where`' part, i.e., middle patterns. Each pattern includes a guard part and a bottom part.

The mapping *cm2Dm* allows to access a set of object pairs of the conceptual model and the design model, which coincide in their name.

In the mapping *assocEnd2Attr*, the first '`check`' part means that when two *ClassCM* objects in the guard part are defined, the restriction including OCL constraints in the bottom part must hold. The second '`check`' part means that when two objects of *ClassDM* in the guard part are defined, two *Attr* objects and two *Op* objects are created and their attribute values are updated by the assignments in the bottom part. The keyword '`realize`' means that the realized

```
class Cm2Dm{
 classCM: ClassCM;
 classDM: ClassDM;
}

class AssocEnd2Attr{
 role: AssocEnd;
 op:Op;
 attr: Attr;
}
```

```
map cm2Dm{
 check cm(){
  classCM: ClassCM;
 }
 check dm(){
  classDM: ClassDM;
 }
 where(){cm2dm:Cm2Dm |
  cm2dm.classCM=classCM;
  cm2dm.classDM=classDM;
  classCM.name
          =classDM.name;
 }}
```

```
map assocEnd2Attr{
 check cm(oneCM:ClassCM; manyCM:ClassCM){
  oneRole:AssocEnd,assoc:Assoc,manyRole:AssocEnd|
  oneRole.class=oneCM; oneRole.assoc=assoc;
  manyRole.class=manyCM; manyRole.assoc=assoc;
  oneRole.multiplicity='1'; manyRole.multplc='*';
  oneRole.roleName<>oclUndefined(String);
  manyRole.roleName<>oclUndefined(String);}
 check enforce dm(oneDM:ClassDM;manyDM:ClassDM){
  realize oneAttr:Attr, realize manyAttr:Attr,
  realize oneOp:Op, realize manyOp:Op |
  oneAttr.class:=manyDM; manyAttr.class:=oneDM;
  oneOp.class:=manyDM; manyOp.class:=oneDM;
  oneAttr.type:=oneDM.name;
  oneOp.retType:=oneDM.name;
  manyAttr.type:=
        'Set('.concat(manyDM.name).concat(')');
  manyOp.retType:=manyAttr.type;}
 where(oneMapping:Cm2Dm, manyMapping:Cm2Dm){
  realize one2Attr:AssocEnd2Attr,
  realize many2Attr:AssocEnd2Attr |
  one2Attr.role:=oneRole;one2Attr.attr:=oneAttr;
  one2Attr.op:=oneOp; many2Attr.role:=manyRole;
  many2Attr.attr:=manyAttr;many2Attr.op:=manyOp;
  oneAttr.name:=oneRole.roleName;
  oneOp.name:='get'.concat(oneRole.roleName);
  manyAttr.name:=manyRole.roleName;
  manyOp.name:='get'.concat(manyRole.roleName)}}
```

Fig. 3. Example Transformation in the QVT Core language

variables must be enforced in order to bind to values. The keyword '`enforce`' means that in case the OCL constraints within the bottom pattern do not hold, the target model, i.e., the design model must be changed to fulfil the constraints.

The transformation executes when its mappings are executed. A combination of the two mappings *cm2Dm* and *assocEnd2Attr* fulfils the requirements stated for the example transformation *addAssociation*.

2.2 TGGs Including OCL for the Example Transformation

Figure 4 presents the rule *addAssociation* in a TGG-like style, but in addition to TGG elements OCL is also used. In the six parts LG, CG, RG, LB, CB, and RB, the items L, C, R, G, B stand for Left, Correspondence, Right, Guard and Bottom, respectively. The notions Guard and Bottom are taken from the QVT Core language. The properties in the lower part of the object rectangles (like *name* or *multiplicity*) indicate object attributes. The TGG rule is a non-deleting rule, i.e., the left-hand side (L=LG+CG+RG) is included in the right-hand side (R=L+LB+CB+RB).

In Fig. 4, OCL constraints for the rule are presented. The OCL constraints cannot only be employed to restrict the attribute values in the guard and bottom part, but they can also express more general restrictions. Classical TGG rules do not embody OCL constraints. Our approach allows to add OCL constraints in all parts and thus enables a general form for restricting the rule application as well as a fine-grained description of consequences of rule application, for example, to describe the updating of object attributes. General OCL expressions, which can traverse the complete working graph on which the rule is applied, are allowed as right sides of attribute assignments. The evaluation of the OCL expressions is made before the working graph is modified by the rule.

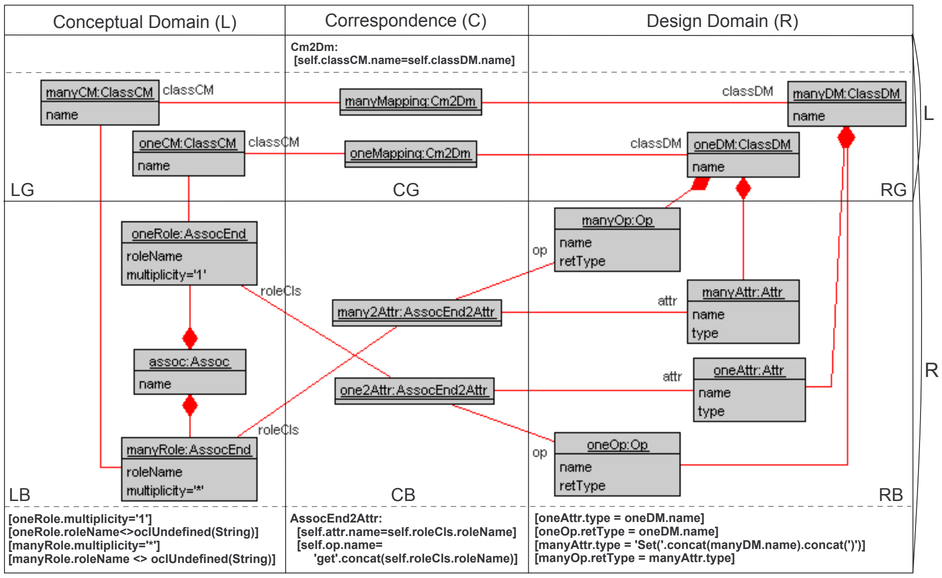

Fig. 4. Example TGG rule including OCL: *addAssociation*

This rule works as follows: Applying the rule within the conceptual domain means to create an *Assoc* object representing an association; this induces a graph rewrite in the design domain, i.e., *Attr* objects and *Op* objects representing the association are created; the correspondence between the added association and the new attributes and operations is established by a graph rewrite in the correspondence domain. The OCL constraints allow to check the applicability of the rule and to update the attribute values within the design domain.

2.3 Requirements for the Integration of TGGs and OCL

By considering QVT as a motivation and a target for the integration of TGGs and OCL as well as by generalizing the results from the subsections 2.1 and 2.2, we propose requirements for the integration of TGGs and OCL as follows.

- Supporting OCL formulas for attribute expressions.
- Supporting OCL conditions as pre- and postconditions of TGG rules (corresponding to negative application conditions and the right-hand side of TGG rules, respectively), which cannot be represented by graphs.
- Supporting constraints on the metamodel of the middle domain of TGG rules. (Supposing that all of the constraints must be taken from TGG rules)
- Supporting bi-directionally executable TGG rules in a synchronous way.

In addition, the integration of TGGs and OCL requires variations of TGGs. A node in the correspondence domain may be connected to multiple nodes in the remaining domains. This is a variation of TGGs because in the original TGGs [1], the mappings are only one-one mappings. With the integration, TGG rules can be deleting rules whereas in the definition of TGGs [1], TGG rules are non-deleting rules.

3 USE Realization of TGGs and OCL

This section presents an approach for the realization of TGG rules including OCL within our tool USE. We propose a language called *use4tgg* for the integration of TGGs and OCL. We obtain operational scenarios of a TGG rule by automatically translating the *use4tgg* description of the TGG rule. The operations are realized in two ways: OCL pre- and postconditions are employed as operation contracts, and command sequences are taken as an operational and executable realization. We extend the technique from previous work [10] in order to realize operational scenarios of a TGG rule.

3.1 Descriptions in the Language use4tgg

Figure 5 presents a *use4tgg* specification for the TGG rule *addAssociation*. Recall that the rule is also represented in Fig. 3 and Fig. 4.

```
rule addAssociation
LG------------------------------------------------------------
  oneCM:ClassCM
  manyCM:ClassCM
LB------------------------------------------------------------
  assoc:Assoc
  oneRole:AssocEnd
  manyRole:AssocEnd
  (assoc,oneRole):Assoc_AssocEnd
  (assoc,manyRole):Assoc_AssocEnd
  (oneCM,oneRole):Class_AssocEnd
  (manyCM,manyRole):Class_AssocEnd
  [oneRole.multiplicity='1']
  [oneRole.roleName<>oclUndefined(String)]
  [manyRole.multiplicity= '*']
  [manyRole.roleName<>oclUndefined(String)]
RG------------------------------------------------------------
  oneDM:ClassDM
  manyDM:ClassDM
RB------------------------------------------------------------
  oneAttr:Attr
  manyAttr:Attr
  oneOp:Op
  manyOp:Op
  (oneDM,manyAttr):Class_Attr
  (manyDM,oneAttr):Class_Attr
  (oneDM,manyOp):Class_Op
  (manyDM,oneOp):Class_Op
  [oneAttr.type=oneDM.name]
  [oneOp.retType=oneDM.name]
  [manyAttr.type='Set('.concat(manyDM.name).concat(')')]
  [manyOp.retType=manyAttr.type]
CG-------------------------------------------------------------
  (oneCM,oneDM) as (classCM,classDM) in oneMapping:Cm2Dm
  (manyCM,manyDM) in manyMapping:Cm2Dm
  Cm2Dm:[self.classCM.name=self.classDM.name]
CB------------------------------------------------------------
  (oneRole,oneAttr) as (roleCls,attr) in one2Attr:AssocEnd2Attr
  (oneRole,oneOp) as (roleCls,op) in one2Attr:AssocEnd2Attr
  (manyRole,manyAttr) in many2Attr:AssocEnd2Attr
  (manyRole,manyOp) in many2Attr:AssocEnd2Attr
  AssocEnd2Attr:[self.attr.name=self.roleCls.roleName]
  AssocEnd2Attr:[self.op.name='get'.concat(self.roleCls.roleName)]
end
```

Fig. 5. TGG rule *addAssociation* in use4tgg.

A TGG rule in *use4tgg* includes six parts. Each part corresponds to a pattern of the TGG rule. Each pair of them corresponds to a domain (left, correspondence, right) rule: (LG, LB), (CG, CB) and (RG, RB). As above, G stands for Guard and B for Bottom. A rule within a domain is specified by the guard and bottom parts (*G, *B). They can be seen as the pre- and postcondition of the rule, respectively.

The language *use4tgg* offers a particular feature for the correspondence domain and allows an extension for specifying patterns of the correspondence. In use4tgg one can specify correspondences between objects of the left and right domains. The syntax for these so-called correspondence links is as follows.

(object_L, object_R) as (role_L, role_R) in object_C: Class_C

The items L, R, and C stand for Left, Right and Correspondence, respectively. The advantage of this extension is that we can automatically generate the metamodel of the correspondence domain. For example, such a correspondence link is employed in the (CG, CB) parts in Fig. 5. From these correspondence links, we can derive the metamodel of the correspondence domain in Fig. 2. In addition, these correspondence links allow to suppress links between the correspondence domain and the left and right domains, i.e., the two links (*object_L*, *object_C*) and (*object_R*,*object_C*) do not have to be given explicitly, because they can be inferred from the correspondence link. Correspondence links highlight the correspondence in a condensed way.

OCL constraints from the correspondence domain (CG, CB) can become constraints (invariants) on the metamodel. For example, OCL constraints in the CG and CB parts in Fig. 5 may be employed for restricting the metamodel in Fig. 2.

3.2 Realization by OCL Pre- and Postconditions

The rule can be realized by OCL pre- and postconditions as an operation contract. We obtain these OCL constraints by applying a sequence of transformation steps on the rule: Initialization, Link Preservation, Link Creation, Link Destruction, Object Creation, Object Preservation and Object Destruction [10].

Figure 6 illustrates the derivation of OCL pre- and postconditions by the transformation steps. The Initialization step is applied on the objects a, b, and d and the links link1 and link2. The Link Preservation step is executed on link2. The Link Creation step creates link3. Then the Link Destruction step is done on link1. The Object Creation step creates the object c. Afterwards, the Object Preservation step is applied on the objects a and b, and the Object Destruction step on the object d. Using these steps, OCL pre- and postcondition are derived. For example, when the Link Destruction step is applied on link1, we obtain the OCL precondition `a.b1→includes(b)` and the OCL postcondition `a.b1→excludes(b)`.

3.3 Realization by Command Sequences

We employ five basic commands: Object Creation, Link Creation, Attribute Assignment, Link Destruction and Object Destruction in order to realize the operation. The commands are illustrated by the part ‘command sequence’ in Fig. 6.

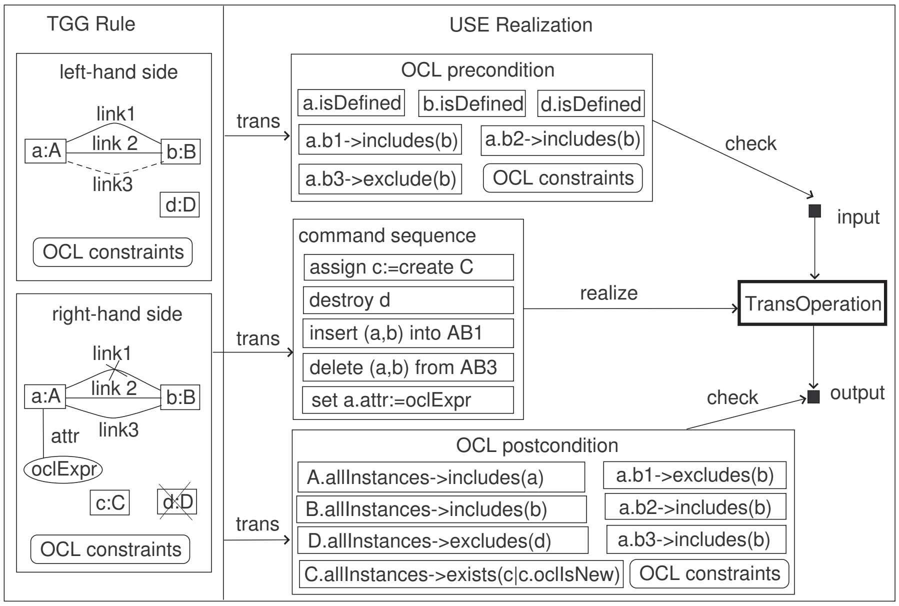

Fig. 6. USE realization for a TGG rule: (i) By imperative command sequences, and (ii) by declarative OCL pre- and postconditions.

Remark 1: Although we do not force to employ assignments in our patterns, certain OCL formulas must be presented in an assignment-like style, if we want to obtain a complete operation realization of a TGG rule in form of a command sequence: Only if in the bottom part of a TGG rule an attribute modification (written down as an OCL formula) is presented in the form `Attribute` = `OclExpression`, the resulting operation is realized in a correct way.

Remark 2: *use4tgg* has two modes for specifying TGG rules. The first mode is indicated with the keyword NONDELETING and the keywords for the rule parts (LG, CG, RG, LB, CB, RB), and the second mode is indicated with the keyword EXPLICIT. The first mode is employed for specifying non-deleting rules, e.g., for treating the example transformation in this paper. This mode is the default and the keyword NONDELETING may be skipped. The second mode is employed for specifying rules which may be deleting. In this mode, all nodes and links must be explicitly stated whereas in the first case only the added nodes and links are mentioned. The pattern format of the two modes is identical. For specifying non-deleting rules, the first mode is much shorter than the second mode.

3.4 Deploying the TGG and OCL Realization in USE

Figure 7 presents the result of applying the transformation *addAssociation*. Snapshot1 is glued with the guard part of the rule, and Snapshot2 is the result of the rule application. These object diagrams represent the abstract syntax of the example, whereas the diagram in Fig. 1 is another representation in concrete syntax.

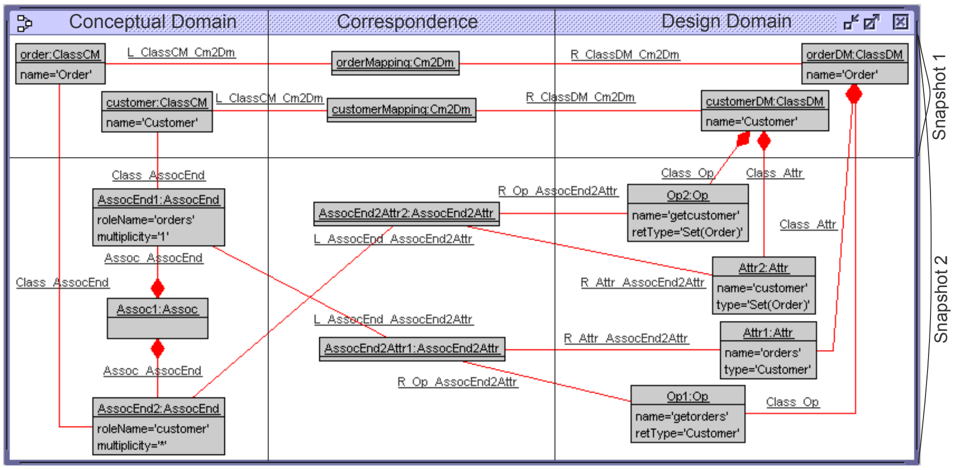

Fig. 7. Applying the TGG rule *addAssociation*: From Snapshot1 to Snapshot2

The integration of TGGs and OCL is completely covered by USE [8]. This is similar to the case for realizing graph transformation within USE [11].

Presenting models: Host models are represented as object diagrams in USE. The models are restricted by the metamodel invariants.

Matching TGG rules: Matching a rule is carried out by evaluating OCL queries on the source object diagram (working graph). These queries are captured by the precondition of the operation corresponding to the rule.

Applying TGG rules: A rule application is realized by an operation call. After each rule application, one may check the postconditions of the rule for an on-the-fly verification of the transformation.

4 Support Model Co-Evolution and Consistency

This section discusses how the integration of TGGs and OCL can support model co-evolution and consistency by means of an example. We want to emphasize again the consistency between a conceptual model and a design model and to maintain the consistency whilst both models are evolving or changed. In exemplary form, Fig. 8 illustrates the situation with the model versions A, B and C. The conceptual (left) and design (right) models are presented in concrete syntax. But the following discussion refers to the models as instances of the UML metamodel.

Co-evolution of models. A co-evolution step between model versions can be represented by a TGG rule including OCL. The left- and right-hand sides of the rule correspond to the target and source versions. The consistency between the models in the different versions is maintained by links and OCL constraints in the correspondence domain of the TGG rule. The co-evolution

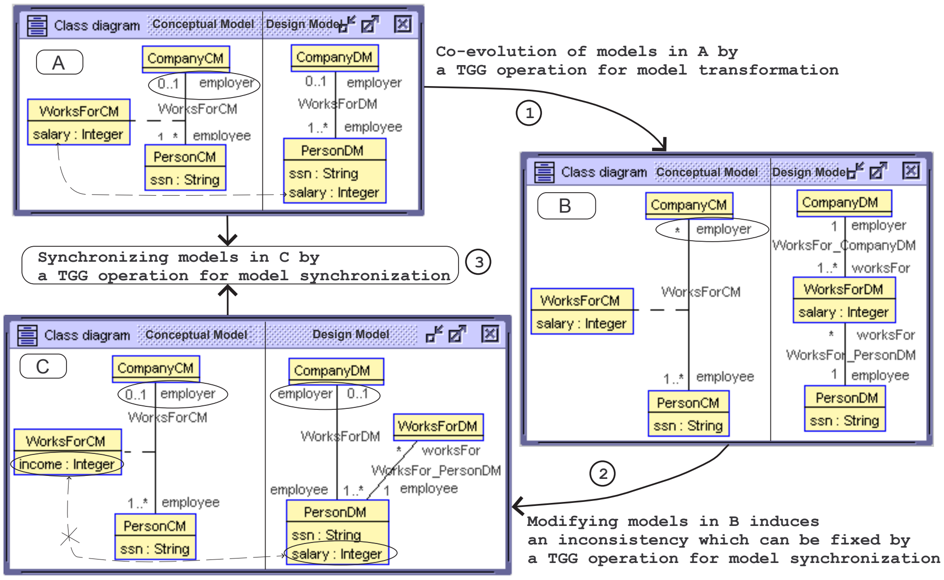

Fig. 8. Model co-evolution and consistency with TGG rules including OCL.

is carried out by a transformation operation derived from the TGG rule: the source version is transformed to the target version. In Fig. 8, see the note (1), a co-evolution from the version A to the version B is carried out by the TGG rule *transAssocclass_one2many* as the association multiplicity in the conceptual model in the version A is changed. The correspondence between the models in the versions A or B expresses constraints for a refinement of an association class in the conceptual model, e.g., names of corresponding Class (Attribute) objects must coincide. In the version A, we recognize the correspondence between two Attribute objects whose *name* attribute is 'salary'. The correspondence (the dashed line in A) is represented by links between the objects and an object Cm2Dm in the correspondence domain of the TGG rule and the OCL invariant in the class Cm2Dm: `self.attrCM.name=self.attrDM.name`.

Detecting inconsistency. We consider an inconsistency in the target version as (Class, Attribute etc.) object attributes are changed. We can detect the inconsistency by checking OCL constraints in the correspondence domain of the TGG rule. In Fig. 8, see the note (2), the version C results from a modification in the version B. In the version C, we recognize that there are two corresponding Attribute objects whose *name* attributes do not coincide because the value 'income' differs from 'salary'. This induces an inconsistency (indicated by the dashed line in C) because the invariant of the class Cm2Dm is not fulfilled: `self.attrCM.name=self.attrDM.name`.

We consider another inconsistency in the target version as links are changed. We can detect the inconsistency in a semi-automatic way by representing models as the image of the TGG rule. We query them from the host models using OCL conditions built by links and OCL constraints in the TGG rule. In Fig. 8, the models in the version C can be queried using OCL conditions derived from the TGG rule $transAssocclass\_one2many$. In the version C, we can recognize an inconsistency since object links are changed inducing that the refinement of the association class is not fulfilled.

Fixing inconsistency. Detecting inconsistency in the way as pointed out above allows us to fix the inconsistency manually. We propose an approach for detecting and fixing automatically the inconsistency in the target version. The inconsistency occurs because of the change of (Class, Attribute etc.) object attributes and links in models of the version. We employ a synchronization scenario of the TGG rule in order to fix the inconsistency. The general aim of the operation is to re-transform the source version to the target version, however, instead of creating objects and links like the co-evolution scenario does, the operation will update objects and links in the target version. In Fig. 8, see the note (3), the versions A and C correspond to the left- and right-hand sides of the TGG rule. The synchronization of the version C allows us to fix the inconsistency. A deeper discussion of our realization of the operation goes beyond the scope of the paper.

5 Related Work

Triple Graph Grammars (TGGs) have been proposed first in [1]. Since then, a lot of further developments and applications have indicated the strength of TGGs in particular within software engineering.

In [12], a comparison between TGGs and the QVT Core language is presented. The paper shows that both approaches are very similar. A difference between them is the use of OCL constraints within the QVT Core language whereas multiple correspondence nodes would do the same within TGG. That paper rises the question how to integrate OCL into TGGs. This can be seen as one motivation for our work.

In [6], a quite complete survey of principles, concepts, usability and implementations of TGGs is discussed. This report proposes a notation for OCL constraints within TGG descriptions. However, OCL constraints within that approach are only for attribute expressions. That paper considers only non-deleting rules and does not go into details about the merits of OCL constraints on metamodels as well as the technical realization for OCL.

The approach in [13] considers the integration of OCL into model transformations within Fujaba. This is realized by integrating the Dresden OCL Toolkit. However, that paper does not mention explicitly TGG concepts as well as OCL constraints in connection with TGGs.

The work in [3] proposes an approach for the integration of a number of MOF-compliant models. So-called Multi-Domain Integration (MDI) rules, a generalization of TGGs, are proposed. The paper mentions the value of declarative specifications in connection with TGGs and OCL. However, the specification of OCL constraints within TGGs is not treated.

In [5, 4, 14], an extension of TGGs for model integration is proposed. The basic idea is that declarative TGG rules may be (semi-)automatically derived in order to obtain operational graph grammar rules for consistency checking, consistency recovery and model transformation. However, within these works, OCL constraints, which are in our view central for such approaches, are not treated.

Our work can be seen an approach to model traceability [15]. The TGG approach supports an interesting feature for the motivation: Trace links can be automatically generated. In [16], the approach to tracing two models by a middle model, a so-called weaving model, is considered. Weaving models are built by weaving metamodels. We can see TGGs as the very transformations on weaving models. OCL constraints within our integration of TGGs and OCL are the very constraints for correspondences within weaving models.

The work in [17] outlines an approach utilizing TGGs for managing consistency between views in general and UML models and semantic domains in particular. In [7], TGGs are employed for model view management. [18] presents a TGG approach for incremental model synchronization. Our work adapts from these papers the intended application scenarios.

Plain textual OCL has been extended to be visualized with collaborations in [19]. Within our previous work [11, 10], we proposed an approach for the realization of plain graph transformation in USE. The current work extends this approach for treating TGGs.

6 Conclusion and Future Work

This paper proposes a language for the integration of Triple Graph Grammars (TGGs) and the Object Constraint Language (OCL). The approach views operational scenarios of a TGG rule as operations and characterizes the operations by OCL pre- and postconditions and by an executable command sequence. The language is realized in our UML and OCL tool USE. USE supports full OCL and UML (metamodel) class diagrams. Thus, metamodels and their transformations can be inspected and analysed in USE. This paper proposes a novel approach for model co-evolution and consistency using the integration of TGGs and OCL.

Future work will include a graphical user interface in combination to the current textual interface for the specification of rules. We aim to extend our language to special QVT features expressed in QVT by keywords as 'check', 'enforce' and 'realize'. A further activity will concentrate on how to generate correspondence metamodels from TGG rules. Middle and large scale case studies must give feedback on the practical applicability of our work. In particular, we see applications in the fundamentals of language engineering where a source language (including syntax and semantics) is translated into a target language (also including syntax and semantics) and where the correspondence domain and the correspondence metamodels take care of correctness criteria.

References

1. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In Mayr, S., ed.: WG, Springer-Verlag, LNCS 903 (1995) 151–163
2. Pratt, T.W.: Pair Grammars, Graph Languages and String-to-Graph Translations. Academic Press **5** (1971) 560–595
3. Königs, A., Schürr, A., Bézivin, J., Heckel, R.: Multi-Domain Integration with MOF and extended Triple Graph Grammars. In: Language Engineering for Model-Driven Software Development, IBFI, Schloss Dagstuhl, Germany, Vol. 04101 (2005)
4. Königs, A.: Model Transformation with Triple Graph Grammars. In Briand, L., Williams, C., eds.: MoDELS, Springer, LNCS 3713 (2005)
5. Königs, A., Schürr, A.: Tool Integration with Triple Graph Grammars - A Survey. ENTCS, Vol. 148 (February 2006) 113–150
6. Kindler, E., Wagner, R.: Triple Graph Grammars: Concepts, Extensions, Implementations and Application Scenarios. Technical Report, University of Paderborn, Germany (2007)
7. Guerra, E., de Lara, J.: Model View Management with Triple Graph Transformation Systems. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G., eds.: ICGT, Springer, LNCS 4178 (2006) 351–366
8. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming (2007)
9. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification ptc/07-07-07. OMG (2007)
10. Büttner, F., Gogolla, M.: Realizing Graph Transformations by Pre- and Postconditions and Command Sequences. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G., eds.: ICGT, Springer (2006) 398–413
11. Gogolla, M., Büttner, F., Dang, D.H.: From Graph Transformation to OCL using USE. In: AGTIVE, Springer (2007)
12. Greenyer, J., Kindler, E.: Reconciling TGGs with QVT. In Engels, G., Opdyke, B., Schmidt, D.C., Weil, F., eds.: MoDELS, Springer, LNCS 4735 (2007) 16–30
13. Stölzel, M., Zschaler, S., Geiger, L.: Integrating OCL and Model Transformations in Fujaba. In Chiorean, D., Demuth, B., Gogolla, M., Warmer, J., eds.: Models/UML Workshop on OCL (OCLApps'2006), ECEASST, Vol. 5 (2006)
14. Grunske, L., Geiger, L., Lawley, M.: A Graphical Specification of Model Transformations with Triple Graph Grammars. In: ECMDA-FA, Springer, LNCS 3748 (2005) 284–298
15. Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model Traceability. IBM Systems Journal **45** (2006)
16. Fabro, M.D.D., Valduriez, P.: Semi-automatic Model Integration Using Matching Transformations and Weaving Models. In Cho, Y., Wainwright, R.L., Haddad, H., Shin, S.Y., Koo, Y.W., eds.: SAC, ACM (2007) 963–970
17. Heckel, R., Küster, J., Taentzer, G.: Towards Automatic Translation of UML Models into Semantic Domains. In Kreowski, H.J., Knirsch, P., eds.: AGT Workshop on Applied Graph Transformation, Grenoble, France, 2002. (2002) 11–22
18. Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: MoDELS, Springer, LNCS 4199 (2006) 543–557
19. Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G.: A visualization of OCL using collaborations. In Gogolla, M., Kobryn, C., eds.: Proc. 4th Conf. UML (2001). Volume 2185 of LNCS., Springer (2001) 257–271