Proc. Int. Conf. on Graph Transformation (ICGT'2006)
Eds. L. Ribeiro, U. Montanari, Springer, Berlin, LNCS (2006).

Realizing Graph Transformations by Pre- and Postconditions and Command Sequences

Fabian Büttner, Martin Gogolla

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

Abstract. This paper studies two realizations of graph transformations which are based on a UML class diagram. The first realization achieves a representation in terms of descriptive pre- and postconditions. The second one yields an operationally executable command sequence in terms of basic commands for object and link creation, attribute modification, and object and link destruction. Our aim for realizing graph transformations in terms of target languages offering different views, i.e., descriptive or operational, is to take advantage of both views and to utilize the benefits which both views provide.

1 Introduction

This paper discusses *model behavior*. Under the notion *model* we understand a collection of UML descriptions [OMG04,RBJ05]. The paper focuses on a special kind of model *behavior*, namely behavior of operations. Behavior of operations is described graphically by a set of graph transformation rules having the aim of achieving understandable and intuitive but strictly formalizable characterizations. These behavior descriptions are *realized* in two different target languages. The first target language is given by OCL pre- and postconditions [WK03]. The second target language is determined by sequences of basic commands for object and link creation, attribute modification, and object and link destruction [OMG04,RBJ05].

We see several advantages in realizing the same source language, in our case graph transformations, in different target languages which possess quite different nature and complement each other: OCL pre- and postconditions have a *descriptive* nature, i.e., it is stated *what* properties have to hold before resp. after operation execution; the command sequences into which we translate have an *operational* nature, i.e., it is expressed *how* the operation is to be executed. Descriptive OCL pre- and postconditions can be analyzed with theorem prover like tools and approaches [BW02,GBR05]. Such tools and approaches aim to deduce new properties from the ones stated explicitly. Operational UML descriptions can be executed and are thus a basis for animating the behavior [OMG04,RBJ05]. Understanding the effect of a complex operation is supported by tracing the single execution steps. Deduction and reasoning as well as animation and prototypical execution is particularly useful is early software development phases.

Realization in different target languages helps to better understand graph transformations insofar that the different target representations emphasize different aspects which are inherently and implicitly present. Realizing graph transformations in the two formalisms can help (1) to formally inspect properties of the graph transformations by checking the achieved OCL constraints with an OCL tool and (2) to understand and test the effects of the graph transformations by executing the achieved command sequences with a UML execution engine. Formal inspection subsumes proving properties of the graph transformation system as a whole.

Our starting point is a set of graph transformations which may be seen as a graphical description of operation behavior in terms of a UML *collaboration*. The UML Language Reference Manual [RBJ05, page 228] explains the notion collaboration as follows:

> A collaboration describes the context for an operation [...] in which the implementation of an operation [...] executes — this is, the arrangements of objects and links that exist when the execution begins, and the [objects and links] that are created or destroyed during execution.

Thus, a UML collaboration involves the objects and links existing before operation execution and the ones existing after operation execution. We distinguish between these two parts of a collaboration and denote a collaboration as a *graph transformation rule* possessing a left side showing the situation before operation execution and a right side presenting the scenario after execution. Rules are common instruments to describe steps within complex processes, and collaborations are the right mechanism to denote them in the UML. In our view, central aspects of graph transformations rules can be expressed in terms of UML diagrams. Utilizing this relationship helps to broaden the audience of graph transformations.

The structure of the rest of this paper is as follows. Section 2 forwards the basic idea of the paper by means of a simple example. Section 3 summarizes the language features in rules which our approach supports. Section 4 discusses the details of the first realization targeting OCL pre- and postconditions. Section 5 shows the details of the second realization targeting command sequences. Section 6 ends the paper with concluding remarks. Both realizations have been prototypically implemented in an extended version of USE [GBR05]. All examples have been checked and executed by the tool.

2 The Basic Idea

Figure 1 shows five example graph transformation rules. The rule fireManager is shown also in the textual rule representation we actually use in our implementation. The rules depend on the upper part of the UML class diagram in Fig. 2 (using the `worksFor` association for representing bi-directional access and the object-valued attribute `staffCar` for uni-directional access). They have a left and right side and name, and offer to employ OCL expressions: (A) In the left side to restrict the rule applicability with a precondition, (B) in the object boxes to modify attributes with OCL expressions (like `aString.value` or

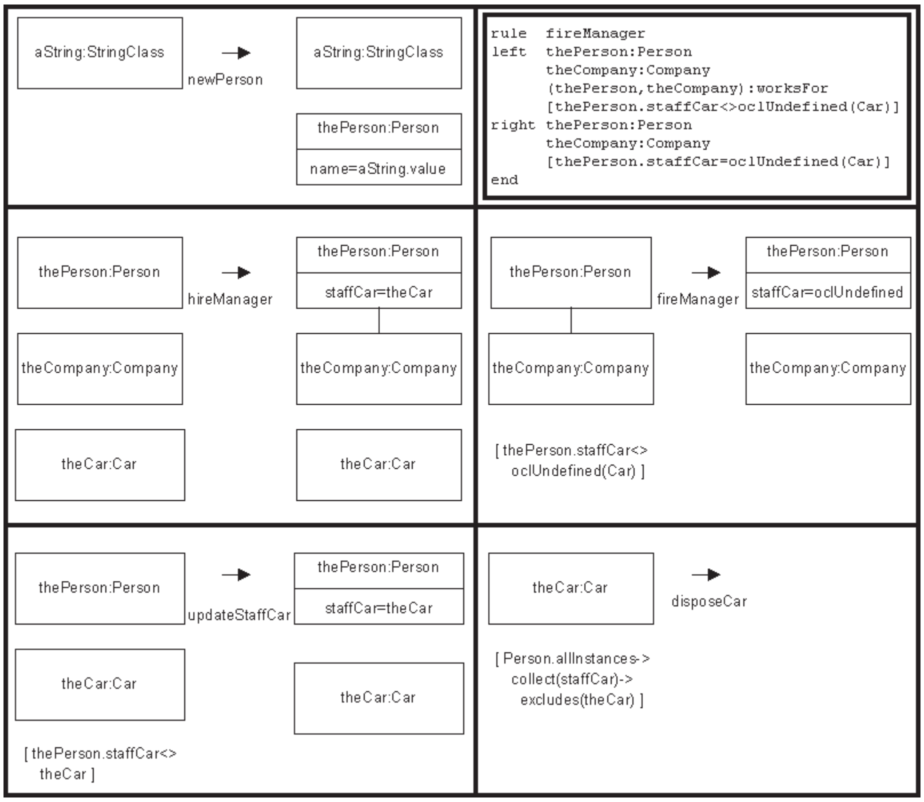

Fig. 1. Example Graph Transformation Rules

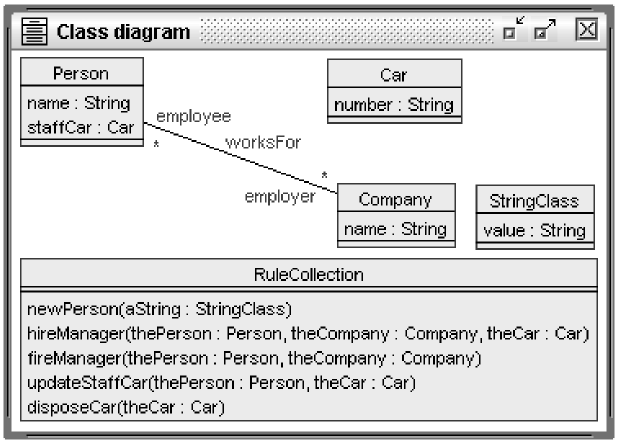

Fig. 2. Example Class Diagram

`theCar`), and (C) in the right side to restrict applicability with a postcondition. This latter feature is not used in the example. The five rules are typical insofar that they cover atomic graph modifications: (A) Create an object, (B) create a link, (C) delete a link, (D) modify an attribute, and (E) delete an object.

```
context newPerson(aString:StringClass)
pre  newPerson_pre: aString.isDefined
post newPerson_post: StringClass.allInstances->includes(aString) and
     Person.allInstances->exists( thePerson | thePerson.oclIsNew and
       thePerson.name=aString.value )

context hireManager(thePerson:Person, theCompany:Company, theCar:Car)
pre  hireManager_pre: thePerson.isDefined and theCompany.isDefined and
     theCar.isDefined and theCompany.employee->excludes(thePerson)
post hireManager_post: Person.allInstances->includes(thePerson) and
     Company.allInstances->includes(theCompany) and
     Car.allInstances->includes(theCar) and
     theCompany.employee->includes(thePerson) and
     thePerson.staffCar=theCar

context fireManager(thePerson:Person, theCompany:Company)
pre  fireManager_pre: thePerson.isDefined and theCompany.isDefined and
     theCompany.employee->includes(thePerson) and
     thePerson.staffCar<>oclUndefined(Car)
post fireManager_post: Person.allInstances->includes(thePerson) and
     Company.allInstances->includes(theCompany) and
     theCompany.employee->excludes(thePerson) and
     thePerson.staffCar=oclUndefined(Car)


-- newPerson(aString:StringClass)
-- assume parameter:Seq(OclAny)=Seq{aString}
let _aString = parameter->at(1);
openter rc newPerson(_aString); assign _thePerson := create Person;
set _thePerson.name := _aString.value;
opexit;

-- hireManager(thePerson:Person, theCompany:Company, theCar:Car)
-- assume parameter:Seq(OclAny)=Seq{thePerson,theCompany,theCar}
let _thePerson = parameter->at(1); let _theCompany = parameter->at(2);
let _theCar = parameter->at(3);
openter rc hireManager(_thePerson,_theCompany,_theCar);
insert(_thePerson,_theCompany) into worksFor;
set _thePerson.staffCar := _theCar;
opexit;

-- fireManager(thePerson:Person, theCompany:Company)
-- assume parameter:Seq(OclAny)=Seq{thePerson,theCompany}
let _thePerson = parameter->at(1); let _theCompany = parameter->at(2);
openter rc fireManager(_thePerson,_theCompany);
set _thePerson.staffCar := oclUndefined(Car);
delete(_thePerson,_theCompany) from worksFor;
opexit;
```

Fig. 3. Conditions and Commands Generated from Graph Transformations

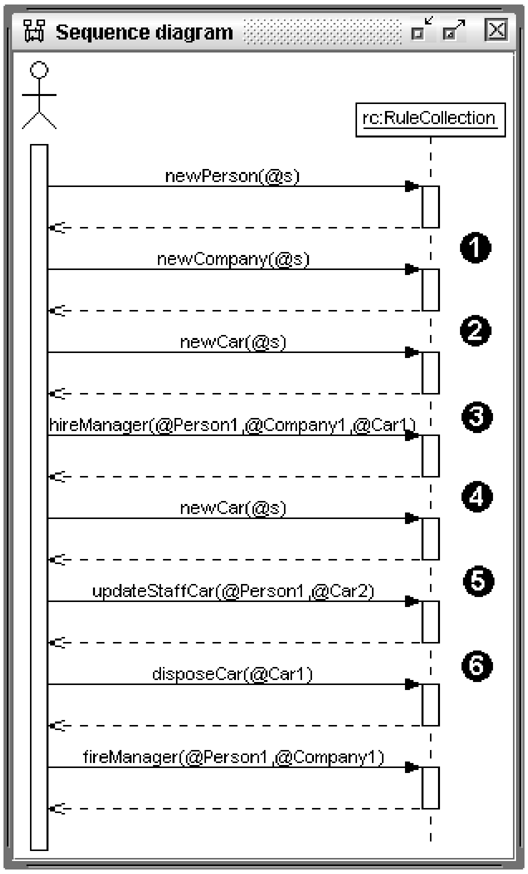

Fig. 4. Example Sequence Diagram

Let us assume that each rule describes one operation. These generated operations are shown in the lower part of Fig. 2 in the class RuleCollection. The rule name determines the operation name, and the left side objects become the operation parameters. We have assigned the operations to a new class, but, if we indicate in the rule to which class the rule belongs, we could distribute the operations to given classes.

The central point we want to make is that we automatically generate an operation specification and an operation implementation from the graph transformations: The upper part of Fig. 3 shows the OCL pre- and postconditions generated for the first three rules, and the lower part of Fig. 3 pictures the generated command sequences for executing the first three graph transformations. Details of the generation process will be discussed in the next section.

Figure 4 shows a sequence of rule applications resp. operation calls in form of a UML sequence diagram. The sequence diagram uses the operations `newCompany` and `newCar` not mentioned before, which are completely analogous to `newPerson`. Figure 5 traces the resulting working graph sequence for these rule applications

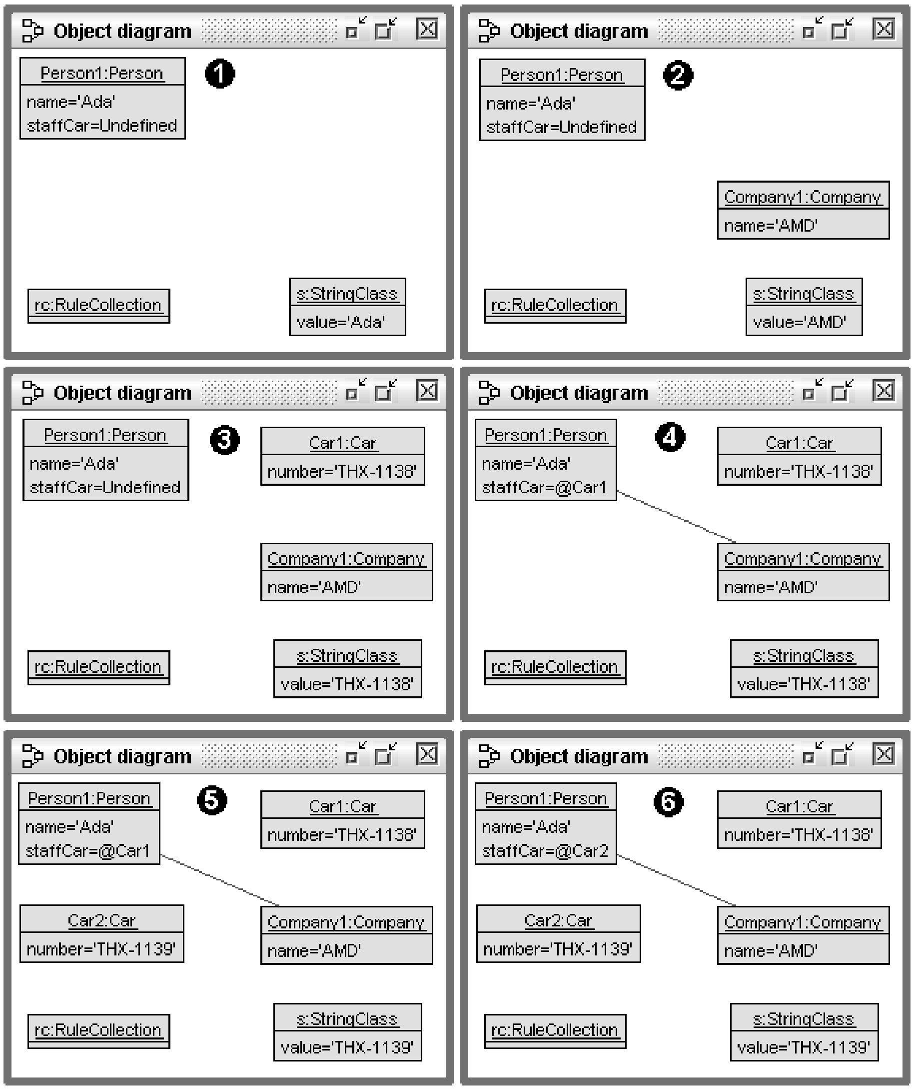

Fig. 5. Example Object Diagrams from Sequence Diagram

with object diagrams. The sequence diagram can be regarded as an abbreviation of a command line protocol where the evaluation of all pre- and postconditions is given in full detail (which we do not show). The form of the arrows in the sequence diagram indicates that (A) the preconditions evaluate to true, i.e., it is allowed to apply the respective graph transformations at the redex determined by the actual parameter, and (B) the postconditions evaluate to true, i.e., the generated implementation with command sequences satisfies its specification. If a pre- or postcondition would have failed, the shape of the arrows would have been different indicating this failure.

To summarize: Our UML tool USE can be employed to execute the graph transformations and can give feedback on their properties. Let us now turn to some details of the conditions and commands in Fig. 3.

Variable 'parameter': Considering the command sequences, each operation is implemented by a command sequence which is called in the sequence diagram. Parameter passing currently works by letting the variable `parameter` hold the sequence of objects which are to be delivered as actual parameters.

Variable 'rc': In the example, we have decided that the new class RuleCollection is the container for the operations. Thus operation calls must be directed to an object `rc` of this class.

Parameters of the Operation: We have said that all objects of the left side become parameters of the operation. We might relax this: The necessity for being a parameter may be dropped, if the parameter is uniquely determined by another parameter, for example, because a *..1 multiplicity is present in the class diagram.

Data Type Valued Parameters: In the example, we had to introduce an auxiliary class StringClass which is essentially there only to pass a String-valued parameter to the operation. If the rule allows to explicitly identify data type valued parameters, such auxiliary classes can be avoided.

Advantage of Using OCL as a Target Language: The advantage that we see in our approach is that it provides full OCL support. For example, at any stage of the execution sequence, the current working graph can be inspected with arbitrary complex OCL queries. Additionally, OCL can be employed in the formulation of graph transformation for expressing rule application restrictions and attribute assignments. Traditional OCL invariants can be incorporated as well. Executing the graph transformation rules, displaying the result with object diagrams, and inspecting the resulting working graph with OCL gives support for checking whether the rules exactly perform the task which the rule designer expected from them.

3 Related Work

Rules have already been successfully applied to (meta-)modeling, for example, in [EHHS00] with a focus on UML 2.0 communication diagrams, but not on OCL on which we concentrate. Our mechanism of executing graph transformation with a UML engine supports checking and validating of graph transformations. We see our approach as an alternative mechanism to existing well tried graph transformation engines like AGG [dLT04], FUJABA [BGN$^+$04], GREAT [KASS03], GROOVE [KR06] or PROGRES [SWZ96]. Our approach is similar to [VFV06] which realizes graph transformations operationally by translating them into basic database commands. [LSE05] proposes to specify contracts for software (pre- and postconditions) in a graphical way with graph transformations. The work concentrates on generating JML (Java Modeling Language) whereas we use OCL offering features not conceptually present in JML, e.g., `allInstances`. [Baa06] discusses in connection with OCL the so-called frame problem dealing with the parts of the system that should remain unchanged when a specific operation is executed. The work proposes a hybrid language with OCL and graph transformation features to handle that problem whereas we keep these two approaches distinct. A constraint language was already part of the graph transformation tool PROGRES. A nice comparison between that language and OCL can be found in [Sch01]. Due to this paper format an in-depth comparison is out of scope.

4 Language Features in Rules

Let us summarize how the rule language which we use for operation description looks like. We explain the language features with the schematic example in Fig. 6. We start with the simplified view that an operation is described by one rule

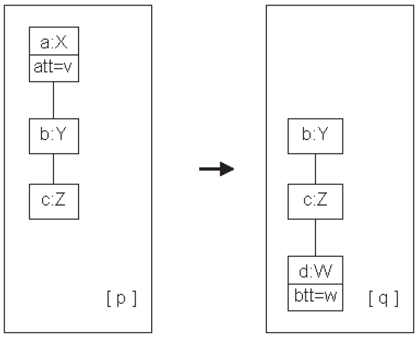

Fig. 6. Language Features in Rules

consisting of a pair of object diagrams where the left side of the rule shows the situation before operation execution and the right side the one after it.

- The connection between the two sides is established through common objects and links occurring in both sides (in Fig. 6 objects b and c and link (b,c)).
- Objects and links may only appear in the left side (in Fig. 6 object a and link (a,b)), whereas other objects and links only show up in the right side (in Fig. 6 object d and link (c,d)).
- The left and the right side may each contain one boolean OCL expression which allows to express an additional rule precondition (in Fig. 6 OCL expression p) and an additional rule postcondition (in Fig. 6 OCL expression q).
- The left side may contain attribute assertions permitting operation execution only if the given attribute values are present (in Fig. 6 attribute assertion att=v with att being an attribute and v being an OCL value expression).
- The right side may contain attribute assignments where the expressions to be assigned are evaluated in the state before operation execution (in Fig. 6 attribute assignment btt=w with btt being an attribute and w being an OCL value expression).

The object diagrams for the left and right side are not proper object diagrams, but they are generic object diagrams in the sense that the objects occurring in the diagram are substituted by concrete objects when the operation is executed and the parameters are passed.

Above we have assumed that one rule describes one operation. This may be generalized: One operation may be characterized by more than one rule without any difficulty if mutually exclusive preconditions are specified in the rules or if mutually exclusive object, link or attribute configurations are stated for different rules of a single operation in the left side. Then the operation can be seen as a larger case distinction which is reflected both in the pre- and postconditions and the command sequences. In future work, we want to allow more powerful operations by allowing for iteration and operation calling in rules as in [ZHG04,GBR05]. Rules would then have similar features like UML 1 collaboration diagrams.

5 Realization by OCL Pre- and Postconditions

The realization of rules by OCL pre- and postconditions is structured into 11 steps: (A) A single step for Initialization, (B) 3 steps handling objects (Object Creation, Object Preservation, Object Destruction), (C) 3 steps handling links (Link Creation, Link Preservation, Link Destruction), (D) 2 steps handling attributes (Attribute Assertion, Attribute Assignment), and (E) 2 steps handling the additional pre- and postcondition (Precondition, Postcondition). Each step treats a particular aspect in the rule and, as depicted in Fig. 7, builds up the OCL description by removing a particular fact from the rule (L,R) and by

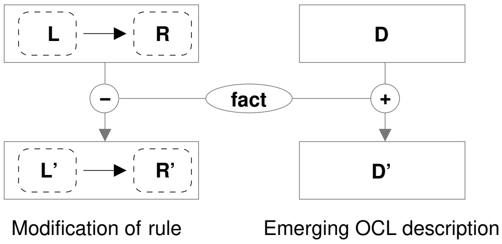

Fig. 7. Structure of Realization Steps for OCL

adding this fact to the OCL description D. By this, one achieves an adapted rule (L',R') and a new OCL description D'. The process stops, if rules are not applicable any more. We show in the following figures the details of the central steps. We will refer to the process of modifying the rule as *rule adaption*.

The order in which the steps have to be applied is restricted as shown in Fig. 8. Within the respective block, the order is not significant, e.g., in the step Object Creation, it does not matter in which order the created objects of the rule are treated.

- The step Initialization in Fig. 9 identifies the objects of the left side as parameters of a newly established operation having a precondition requiring the actual parameters to be different from undefined. The step does not adapt the rule (no rule adaption).

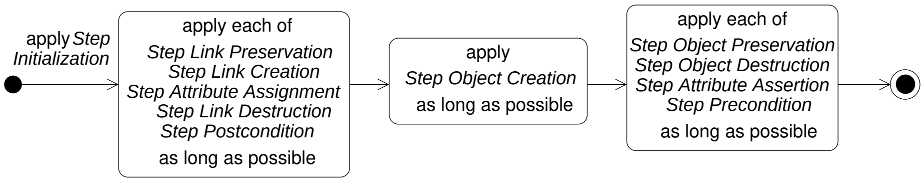

Fig. 8. Order of Realization Steps for OCL

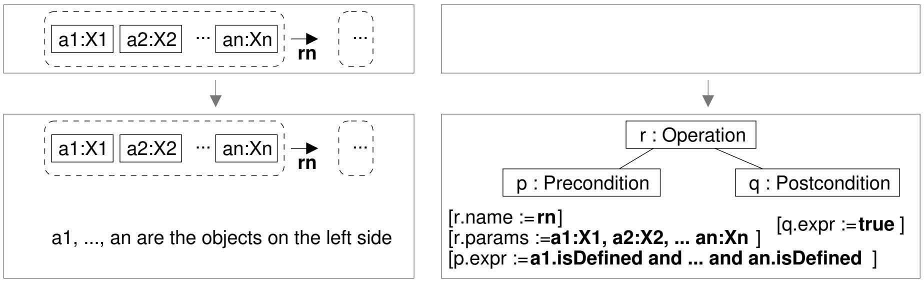

Fig. 9. Step Initialization

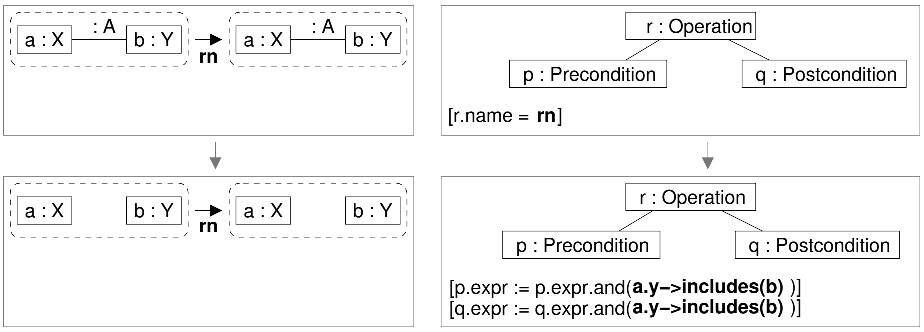

Fig. 10. Step Link Preservation

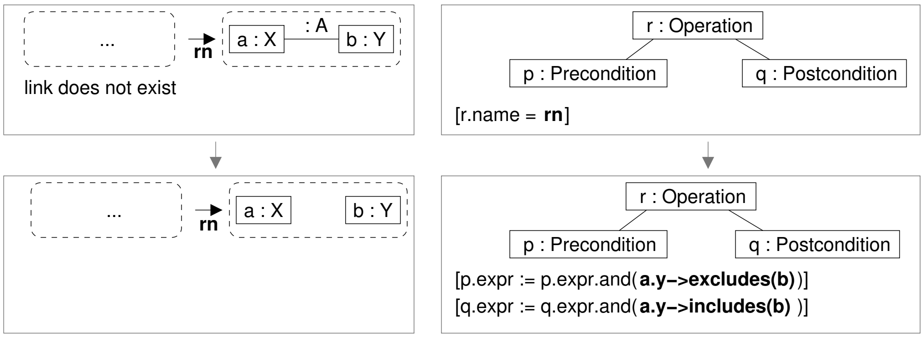

Fig. 11. Step Link Creation

– The step Link Preservation in Fig. 10 extends the already constructed pre- and postcondition by requirements for link existence at the start of the operation as well as at the end of the operation. The step removes the considered link and thus indicates that this link has been completely handled.

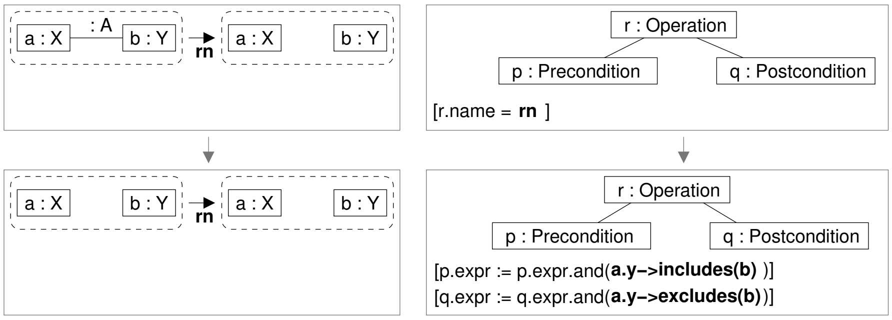

Fig. 12. Step Link Destruction

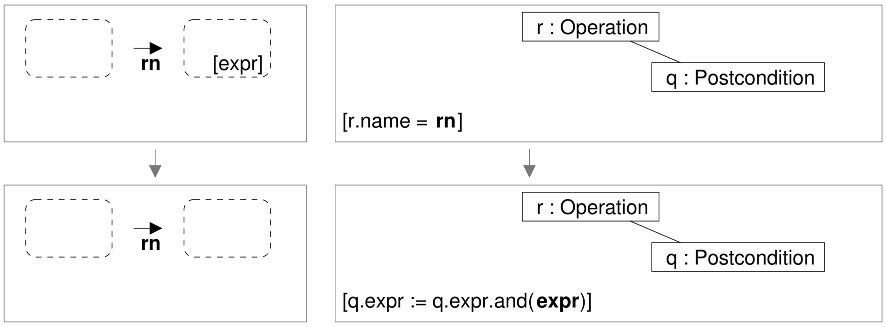

Fig. 13. Step Postcondition

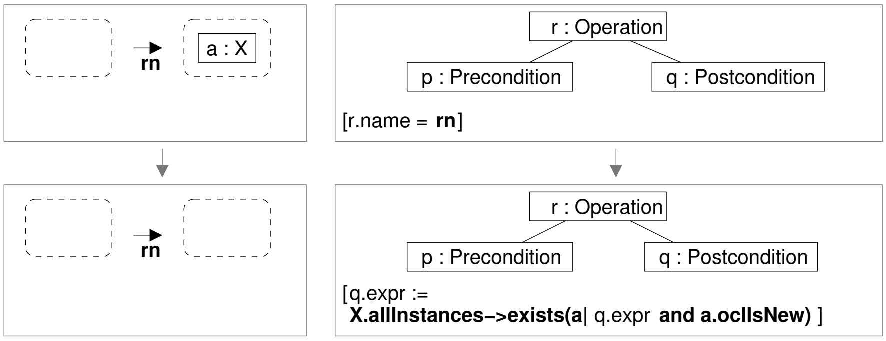

Fig. 14. Step Object Creation

- The step Link Creation in Fig. 11 guarantees in the precondition that the link is not existent and in the postcondition that the link exists. The step removes the link (rule adaption).
- The step Link Destruction in Fig. 12 guarantees in the precondition the existence of the link and in the postcondition the link's non-existence. The step removes the link (rule adaption).
- The step Postcondition in Fig. 13 adds the given OCL expression to the postcondition and removes the OCL expression.
- The step Object Creation in Fig. 14 guarantees in the postcondition that a new object is created and that the postcondition constructed so far is valid for this new object. The step removes the object (rule adaption).

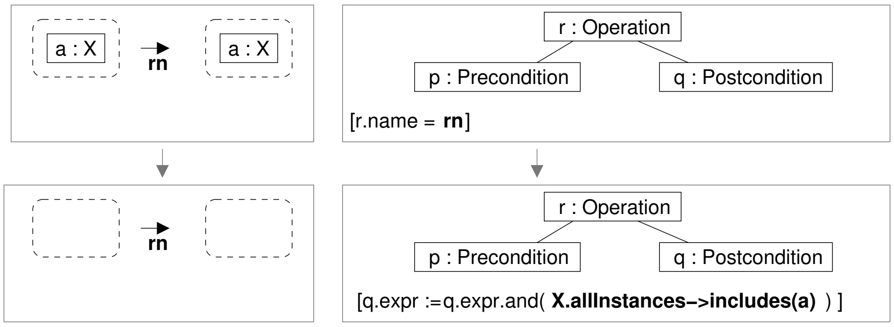

Fig. 15. Step Object Preservation

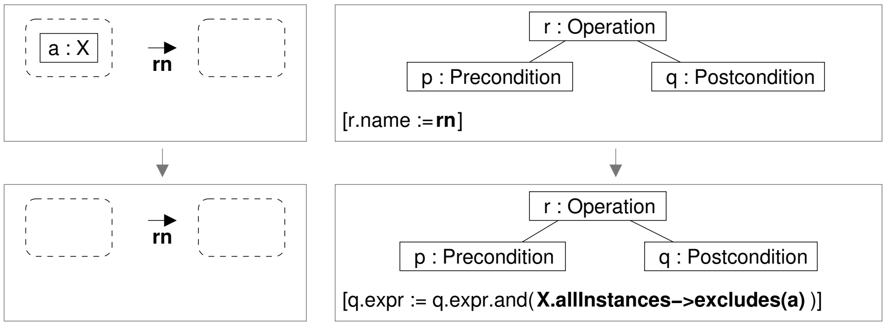

Fig. 16. Step Object Destruction

- The step Object Preservation in Fig. 15 introduces the requirement that a preserved object still exists at postcondition time. Please recall, that all left side objects become operation parameters and are required in the initialization step to be defined, i.e., to be existent. The step removes the object (rule adaption).
- The step Object Destruction in Fig. 16 requires in the postcondition that the object does not belong to the current objects any more. The step removes the object (rule adaption).

We do not show the details of the steps Precondition, Attribute Assertion, and Attribute Assignment. Precondition works analogously to step Postcondition, Attribute Assertion is similar to step Precondition (because an assertion is basically a precondition), and Attribute Assignment resembles step Precondition (because an assignment is basically a postcondition).

6 Realization by Command Sequences

The realization of the rules by command sequences is structured into 5 steps as pictured in Fig. 17. The 5 steps and their order are as follows.

- Object Creation: The objects introduced by the right side of the rule are created.

- Link Creation: The links introduced by the right side of the rule are created by insertion into the link set of the association.
- Attribute Assignment: The attributes are modified according to the details given in the rule.
- Link Destruction: The links occurring only in the left side of the rule are destroyed by removing them from the link set of the association.
- Object Destruction: The objects occurring only the left side of the rule are destroyed.
- Steps *Link Creation* and *Link Destruction* may alternatively and equivalently be realized by modifying an object- and set-valued attribute on either side of the association (Steps 2b and 4b).

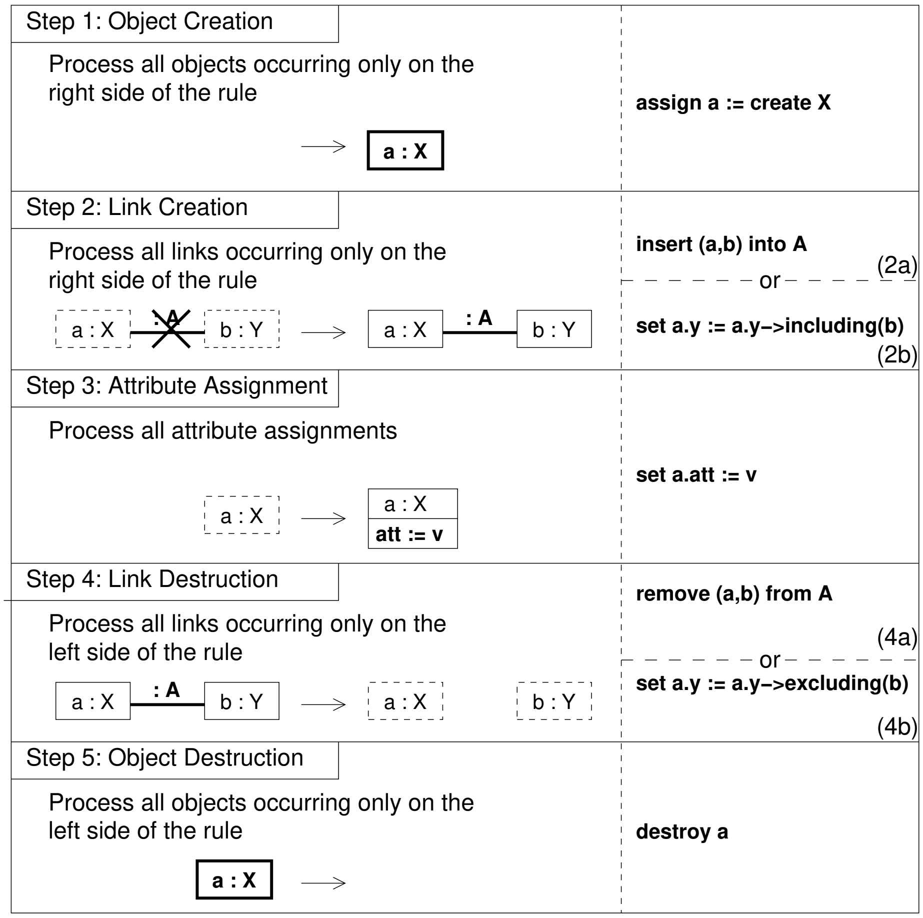

Fig. 17. Realization Steps for Command Sequence Generation

We do not cover handling of rule pre- or postconditions or attribute assertions. Basically, these features can be realized by allowing conditional execution of the commands already shown.

7 Conclusion

This paper proposes to translate graph transformations rules into two different formalisms: (A) A descriptive language expressing the effect of a rule in form of pre- and postconditions, and (B) a procedural language describing the effect of a rule in terms of basic imperative manipulation commands. The procedural realization can be employed to execute the graph transformations. The descriptive realization can be checked against the procedural one, i.e., the pre- and postconditions are evaluated during the execution. The pre- and postconditions can further be used for deduction and verification purposes. Both realizations give feedback to graph transformation designers for their development process. This feedback is in particular needed when the graph transformations work on a metamodel, e.g., the UML metamodel. We think that translating graph transformation into a standard software engineering language like UML including OCL broadens the audience of graph transformations, because standard UML and OCL tools are familiar for many developers. We have made double use of graph transformation rules: Firstly, we have used rules as the source language which we translate and realize in the two formalisms; secondly, we have described our translation in form of graph transformations.

We have used OCL as a vehicle, but we think that the basic idea behind the translation is independent of OCL. We could have used other languages like SQL achieving similar results.

Our approach enables us to automatically compute for a given set of rules and a given working graph, the redexes where rules could be applied. Multiplicities in the underlying class diagram can help to dramatically reduce the search space for the redexes. Due to space limitations we have not described this in detail. Other future work includes the following topics.

- There are several transformation alternatives to the decisions we have taken. For example, instead of using `oclIsNew` we could use explicitly `Class.allInstances->excludes` and `includes` in the pre- resp. postcondition. By this we would reduce the number of used OCL features. Another variation point concerns the parameters of the generated operations, which not necessarily have to be all objects of the left side. But the parameter list should be reduced only if the redex afterwards still uniquely determines the rule application.
- We want to improve the expressiveness of operation descriptions by allowing iteration and operation calling in the rules and plan to group rules into larger structuring units, so-called transformation units.
- A fundamental treatment of the relationship between the generated pre- and postcondition pair and the generated command sequence is needed, i.e., a study of the relationship between the descriptive and the operational view on rules. Such a study would ideally be inspired by the equivalence relationships between relational algebra and relational calculus from the database field.

References

[Baa06] T. Baar. OCL and Graph-Transformations: A Symbiotic Alliance to Alleviate the Frame Problem. In MoDELS 2005 Satellite Events: International Workshops, Springer, Berlin, LNCS, pp. 20-31, 2006.

[BGN+04] S. Burmester, H. Giese, J. Niere, M. Tichy, J.P. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool integration at the meta-model level: the Fujaba approach. *STTT*, 6(3):203–218, 2004.

[BW02] A.D. Brucker and B. Wolff. HOL-OCL: Experiences, consequences and design choices. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *Proc. 5th Int. Conf. Unified Modeling Language (UML'2002)*, 196–211. Springer, LNCS 2460, 2002.

[dLT04] J. de Lara and G. Taentzer. Automated Model Transformation and Its Validation Using AToM 3 and AGG. In A.F. Blackwell, K. Marriott, and A. Shimojima, editors, *Diagrams*, LNCS 2980, 182–198. Springer, 2004.

[EHHS00] G. Engels, J.H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000)*, 323–337. Springer, LNCS 1939, 2000.

[GBR05] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Journal on Software and System Modeling*, 4(4): 386-398, 2005.

[KASS03] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *Journal Universal Computer Science*, 9(11):1296–1321, 2003.

[KR06] H. Kastenberg and A. Rensink. Model Checking Dynamic States in GROOVE. In A. Valmari, editor, *SPIN*, LNCS 3925, 299–305. Springer, 2006.

[LSE05] M. Lohmann, S. Sauer, and G. Engels. Executable Visual Contracts. In IEEE Symp. Visual Languages and Human-Centric Computing (VL/HCC'05), 63-70, 2005.

[OMG04] OMG, editor. *OMG Unified Modeling Language Specification, Version 2.0.* OMG, 2004.

[RBJ05] J. Rumbaugh, G. Booch, and I. Jacobson. *The Unified Modeling Language Reference Manual, Second Edition.* Addison-Wesley, Reading, 2005.

[Sch01] A. Schürr. Adding Graph Transformation Concepts to UML's Constraint Language OCL. In Proc. ETAPS Workshop UNIGRA, ENTCS, 44 (4), 403-410, 2001.

[SWZ96] A. Schürr, A.J. Winter, and A. Zündorf. Developing Tools with the PROGRES Environment. In M. Nagl, editor, *IPSEN Book*, LNCS 1170, 356–369. Springer, 1996.

[VFV06] G. Varró, K. Friedl, and D. Varró. Implementing a Graph Transformation Engine in Relational Databases. *Journal on Software and System Modeling*, 2006.

[WK03] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 2003. 2nd Edition.

[ZHG04] P. Ziemann, K. Hölscher, and M. Gogolla. From UML Models to Graph Transformation Systems. In M. Minas, editor, *Proc. Workshop Visual Languages and Formal Methods (VLFM'2004).* ENTCS, 127(4), 17-33, 2004.