UML Interaction Diagrams: Correct Translation of Sequence Diagrams into Collaboration Diagrams*

Björn Cordes, Karsten Hölscher, and Hans-Jörg Kreowski

University of Bremen, Department of Computer Science
P.O. Box 330440, D-28334 Bremen, Germany
{bjoernc,hoelsch,kreo}@informatik.uni-bremen.de

Abstract. In this paper, the two types of UML interaction diagrams are considered. A translation of sequence diagrams into collaboration diagrams is constructed by means of graph transformation and shown correct.

1 Introduction

The Unified Modeling Language (UML) is a graphical object-oriented modeling language used for the visualization, specification, construction, and documentation of software systems. It has been adopted by the Object Management Group (OMG) and is widely accepted as a standard in industry and research (cf., e.g., [2],[17]). The UML provides nine types of diagrams for different purposes. This paper focuses on sequence and collaboration diagrams collectively known as interaction diagrams. Both diagram forms concentrate on the presentation of dynamic aspects of a software system, each from a different perspective. Sequence diagrams stress time ordering while collaboration diagrams focus on organization. Despite their different emphases they share a common set of features. Booch, Rumbaugh, and Jacobson claim that they are semantically equivalent since they are both derived from the same submodel of the UML metamodel, which gives a systematic description of the syntax and semantics of the UML.

In this paper we provide some justification of this statement by presenting a correct translation of sequence diagrams into collaboration diagrams by means of graph transformation. As illustrated in Figure 1, the translation consists of two steps, each modeled as a graph transformation unit. In the first step a sequence diagram is translated into a metamodel object diagram. The metamodel object diagram is then translated into a corresponding collaboration diagram. In order to accomplish these tasks, the diagrams involved are considered and represented as labeled and directed graphs in a straightforward way where we use additional node and edge attributes during the translation. In this way, the two translations

* Research partially supported by the EC Research Training Network SegraVis and the DFG project UML-AID.

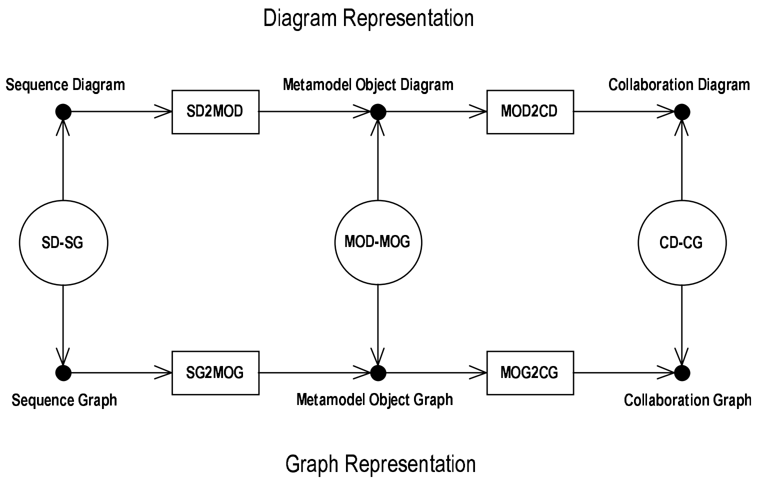

Fig. 1. Overview of the translation

on the level of diagrams are obtained by the two translations on the graph level and going back and forth between diagrams and corresponding graphs. The translation focuses on sequence diagrams on instance level with synchronous and nonconcurrent communication restricted to procedure calls. The diagrams may contain conditions, iteration conditions for procedure calls, and conditional branchings. A complete coverage of all features described in the UML is beyond the scope of this paper.

Our translation contributes to the on-going attempt to develop a formal semantics of UML based on graph transformation (cf. [6], [7], [8], [12], [13], [15], [16], [18]).

The paper is organized in the following way. In the next section, the basic notions of graph transformation are recalled as far as needed. Section 3 provides the translation of sequence graphs while in Section 4 sequence graphs are specified. A short example translation is presented in Section 5. In Section 6 we discuss how the translation can be proved correct, which is followed by some concluding remarks.

2 Graph Transformation

In this section, the basic notions and notations of graph transformation are recalled as far as they are needed in this paper.

In general, graphs consist of nodes and edges, where each edge has a source and a target node. Two different kinds of labels are assigned to the parts (i.e. nodes and edges) of a graph. The first kind of labels are fixed ones over a given alphabet, providing a basic type concept. The second kind are mutable labels, that are used as attributes. An attribute is regarded as a triple comprising the name of the attribute, its type and its current value. While the assignment of fixed labels to the parts is unique, there can be more than one attribute assigned to them.

The concept of graph transformation has been introduced about thirty years ago as a generalization of Chomsky grammars to graphs. Local changes in a graph are achieved by applying transformation rules. The graph transformation approach used in the context of this paper is based on the single pushout approach (cf., e.g., [5]). A formal embedding of the attribute concept into the single pushout approach can be found in [14]. A transformation rule consists of a left-hand side and a right-hand side, both of which are graphs. They share a common subgraph, which is called application context. In examples, the letter **L** indicates left-hand sides and **R** right-hand sides. The application context consists of all nodes of **L** and **R** with the same numbers. A rule can be applied to a given host graph if a match of the left-hand side of the rule in the host graph can be found. A match is a graph morphism which respects the structure of the involved graphs and the labels of their parts. The application of the rule to a host graph is achieved by adding those parts of the right-hand side that are not part of the application context to the host graph. Afterwards the parts of the left-hand side that are not part of the application context are deleted. The resulting structure is then made a graph by removing possible dangling edges. In this approach a rule can always be applied if a match is found, and in case of conflicting definitions deletion is stronger than preservation.

Note that it is possible to demand a concrete value of an attribute in the left-hand side of a rule. If the application of a rule is meant to change an attribute depending on its current value (e.g. increase a value by one), that value has to be identified as a variable in the left-hand side. The operation that is specified in the right-hand side of the rule then has to be performed on the value represented by that variable. If attributes are omitted in a rule, their values are of no interest. Thus attributes are only specified if one wants a concrete value to be present for the rule application or if that value has to be changed, either by calculating a new value or by simply overwriting the current value with a new one.

The left-hand side of a rule describes a positive application condition in the sense that its structure has to be found in the host graph. Sometimes it may be necessary to define a situation that is **not** wanted in the host graph. As proposed in [5], a situation that is not wanted in the host graph can be depicted as an additional graph called constraint. It shares a common subgraph with the left-hand side of the rule. If a match of the left-hand side into the host graph is found, and this match can be extended to map the constraint totally into the host graph, the rule must not be applied. If no such extension of the match can be found, the match satisfies the constraint and the rule can be applied. A rule can have a finite set of constraints, which is called negative application condition. In examples, the letter **N** indicates constraints (where indices distinguish different ones of the same rule). Nodes that are common in left-hand sides and constraints are numbered equally. A match satisfies a negative application condition if it satisfies all its constraints. Thus a rule with a negative application condition

NAC may only be applied if the match satisfies NAC. In [5], the left-hand side of a rule is required to be a subgraph of each constraint. This can always be met in our case by adding the missing parts to the constraints.

As a structuring principle, the notion of a graph transformation unit is employed (cf., e.g. [10],[11]). Such a transformation unit is a system tu $=$ (I, U, P, C, T) where I and T are graph class expressions specifying sets of initial and terminal graphs respectively, U is a set of (names of) imported transformation units, P is a set of local rules, and C is a control condition to regulate the use of the import and the application of the rules. In examples, we specify the five components after the respective keywords *init*, *uses*, *rules*, *cond*, and *term*. A keyword is omitted in case of a default component where *all* (graphs) is the default graph class expression, the empty set the default import and the default rule set, and *true* (allowing everything) the default control condition. The further graph class expressions and control conditions we use are explained when they occur the first time.

3 Translation

In this section, the translation of sequence graphs into collaboration graphs is modeled in a structured and top-down fashion by means of graph transformation units. It is beyond the scope of this paper to present the translation in all details. We introduce explicitly the structure of the translation and some significant transformation units, but we omit units which are defined analogously to given ones. Readers who want to see the complete specification are referred to [3].

The main unit splits the translation into two steps translating sequence graphs into metamodel object graphs first and then the latter ones into collaboration graphs. Moreover, it states that the translation is initially applied to sequence graphs generated by the transformation unit **GenerateSG**, which is specified in the next section.

SG2CG
init: **GenerateSG**
uses: **SG2MOG**, **MOG2CG**
cond: **SG2MOG**; **MOG2CG**

The control condition is a regular expression requiring that the two imported units are applied only once each in the given order.

Both the imported units enjoy essentially the same basic structure each except that all graphs are accepted as initial. In both cases, the collaboration parts and the interaction parts are handled separately followed by deleting graph components that are no longer needed. In the first unit, there is also an additional initial step.

SG2MOG
uses: **initMOG**, **SG2MOG-Collaboration**, **SG2MOG-Interaction**, **DeleteSG**
cond: **initMOG**; **SG2MOG_Collaboration**; **SG2MOG_Interaction**; **DeleteSG**

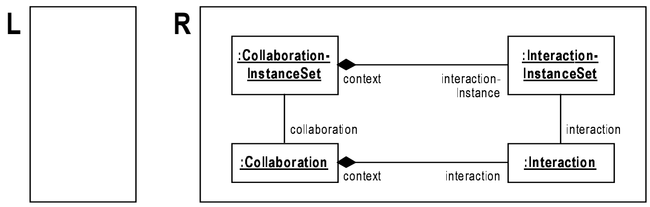

Fig. 2. Rule initMOG-1: Generating the initial metamodel object graph

MOG2CG
uses: **MOG2CG-Collaboration**, **MOG2CG_Interaction**, **DeleteMOG**
cond: **MOG2CG-Collaboration**; **MOG2CG_Interaction**; **DeleteMOG**

Altogether, the translation consists of seven sequential steps which are quite similarly structured to each other. None of them imports other units, but each consists of local rules only which are applied in a certain order given by the respective control conditions. Whenever we refer to UML features in the following units, their initial letters are printed in capitals as in the UML documents.

Let us start with the unit **initMOG**.

initMOG
rules: initMOG-1
cond: initMOG-1

It consists of a single rule which is applied only once. It adds an initial metamodel object graph disjointly to the given graph (see Figure 2).

Let us consider now the unit **SG2MOG-Collaboration**. It has got six rules, named sg2mog-C1a, -C1b, -C2a, -C2b, -C2c, and -C3. The rules are explicitly given in Figures 3 and 4. The control condition requires that either sg2mog-C1a or sg2mog-C1b is applied first, then either sg2mog-C2a or sg2mog-C2b is applied as long as possible, and afterwards sg2mog-C2c is applied once. This sequence is then repeated as long as possible, and finally sg2mog-C3 is applied once.

SG2MOG-Collaboration
rules: sg2mog-C1a, sg2mog-C1b, sg2mog-C2a, sg2mog-C2b, sg2mog-C2c, sg2mog-C3
cond:((sg2mog-C1a|sg2mog-C1b);(sg2mog-C2a|sg2mog-C2b)!;sg2mog-C2c)!;sg2mog-C3

The rules sg2mog-C1a and sg2mog-C1b add a new ClassifierRole, where sg2-mog-C1b adds a new confirming Object in addition. The regarded object is also marked with a writeClassifiers loop inidicating that all of its classifiers will be added next. The negative application conditions ensure, that neither the ClassifierRole nor the Object to be added already exists in the metamodel object graph. The rules sg2mog-C2a, sg2mog-C2b, and sg2mog-C2c concern base Classifiers. While sg2mog-C2a adds a new one, sg2mog-C2b links a role to an existing one. And sg2mog-C2c ends the addition. Finally, sg2mog-C3 inserts a first Message of the Interaction, with the ClassifierRole of the SGObject identified by the Current loop as both sender and receiver.

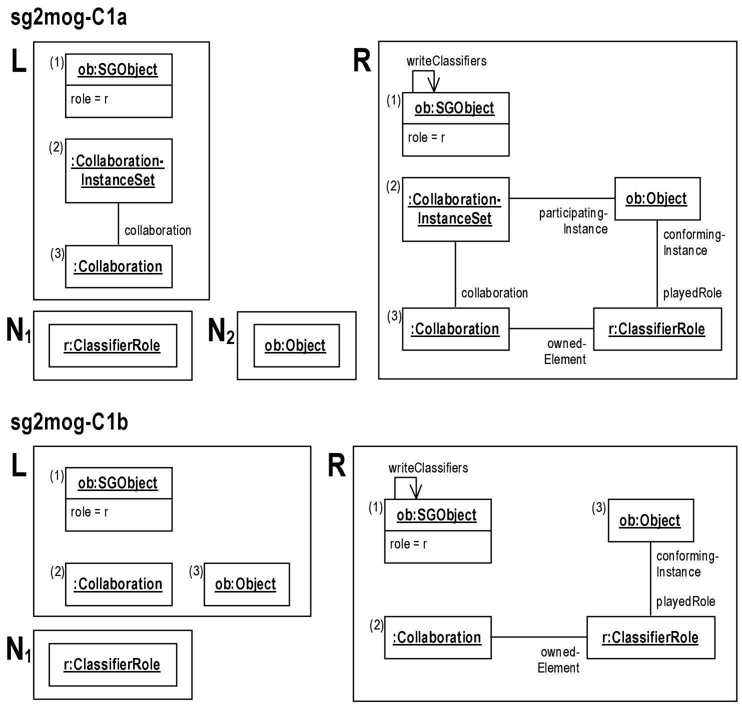

Fig. 3. Rules sg2mog-C1a and sg2mog-C1b

The control condition is formally described by a generalized regular expression. The symbol ';' denotes the sequential composition and the symbol '|' the alternative choice. The symbol '!' is similar to the regular Kleene star '*'. But while the latter one allows the repetition of the preceding expression as long as one likes, the exclamation mark requests repetition as long as possible. Therefore, the condition above allows to add as many ClassifierRoles and Objects as needed where each role is connected to all its base Classifiers and a first Message of the Interaction is inserted (which closes the role addition). The further transformation units used by **SG2MOG** are only sketched without the explicit rules.

The transformation unit **SG2MOG-Interaction** translates the interaction. The rules are designed to process the sequence graph in a certain order, thus a control condition is not necessary. The ordered translation is achieved by using a loop of type Current, that is moved from one node to the next node during the process. Terminal graphs contain a node that is incident with both an End and a Current loop. The class of all these graphs is denoted by **Current&End**.

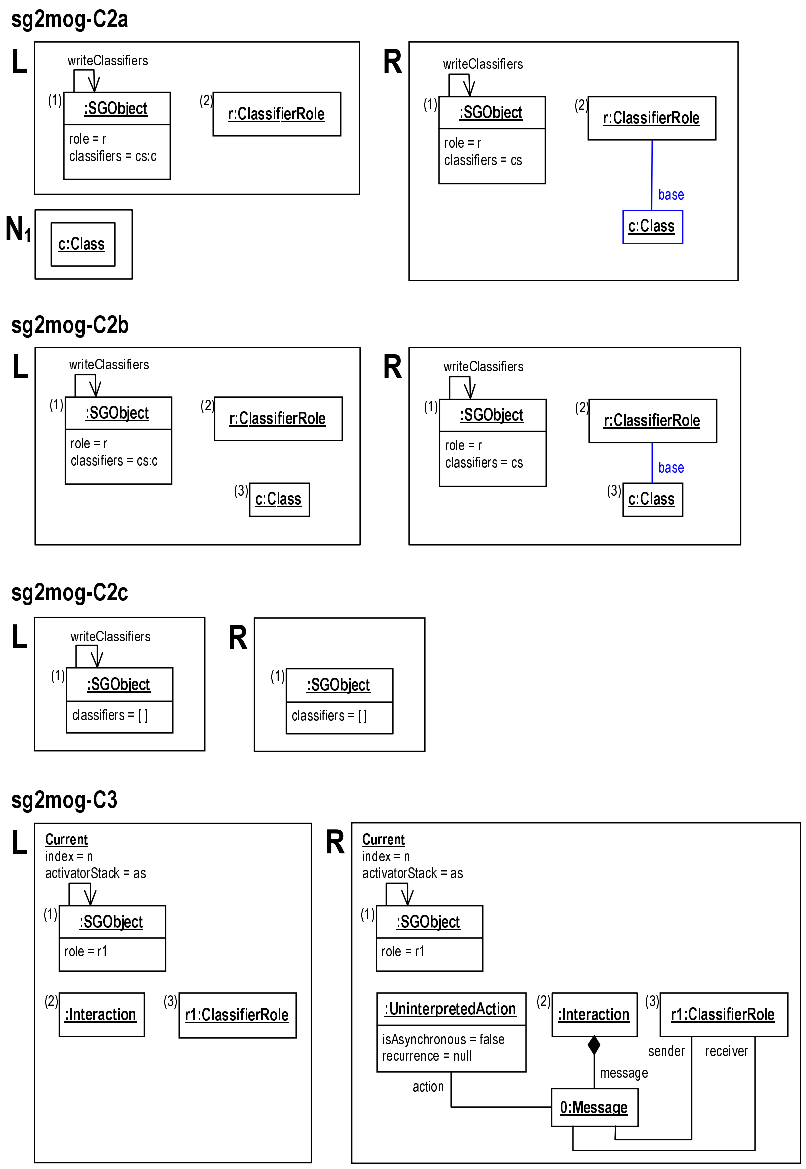

Fig. 4. Rules sg2mog-C2a, sg2mog-C2b, sg2mog-C2c, and sg2mog-C3

SG2MOG-Interaction
rules: sg2mog-I1a, sg2mog-I1b, sg2mog-I2a, sg2mog-I2b, sg2mog-I3a, sg2mog-I3b, sg2mog-I4a, sg2mog-I4b, sg2mog-I5a, sg2mog-I5b
term: **Current&End**

The rules sg2mog-I1a and -I1b add a stimulus to the metamodel object graph, where sg2mog-I1a translates a stimulus with a predecessor and sg2mog-I1b one without a predecessor. Returns are translated in a similar way by the rules sg2mog-I2a and -I2b. The rule sg2mog-I3a prepares the translation of a conditional branching while sg2mog-I3b simply moves the Current edge along the Activation edge to the next Object node. The rules sg2mog-I4a and -I4b translate a branch of a conditional branching with or without predecessor. Rule sg2mog-I5a completes the translation of one branch, while rule sg2mog-I5b completes the whole conditional branching.

Applying the transformation units **SG2MOG-Collaboration** and **SG2MOG-Interaction** yields the desired metamodel object graph. The sequence graph is no longer needed, so it is deleted using the transformation unit **DeleteSG**. This unit simply removes all the nodes of the sequence graph (and all of its edges with them).

Because of lack of space, we omit the specification of **MOG2CG**, which looks similar to **SG2MOG** anyway.

4 Generation of Sequence Graphs

The sequence graphs that can be translated into collaboration graphs by the transformation unit **SG2CG** are generated by **GenerateSG**.

GenerateSG
init: SG_0
rules: gsg1, gsg2, gsg3

The initial graph and the rules are explicitly given in the Figures 5 and 6. The initial graph SG_0 consists of two nodes of type SGObject that are connected by an Activation edge. The source node of this edge is marked as Current with index 1, and 0 the only element on the activatorStack. The target node is marked with an End edge. The values of the name, role and classifier attributes of the nodes can be freely chosen. Since both nodes are on the activation path they represent the same Instance. Thus the attribute values of both nodes must be equal.

Rule gsg1 is used to insert an SGStimulus with a matching Return in an activation path. In the left-hand side of the rule the two nodes of that activation path are identified. In order to ensure synchronous communication, the negative application graph N_1 secures that node 2 cannot be the target node of a Return edge. This means that the Instance represented by the identified nodes cannot send another procedure call while the current one is not yet completed.

The negative application graph N_2 prohibits node 1 from being the target node of a Return edge. This condition enforces that sequences are built from back to front.

The right-hand side of the rule adds two SGObject nodes to the activation path of the two identified nodes. These determine the attribute values of the newly added nodes. The first node (concerning the order in the activation path)

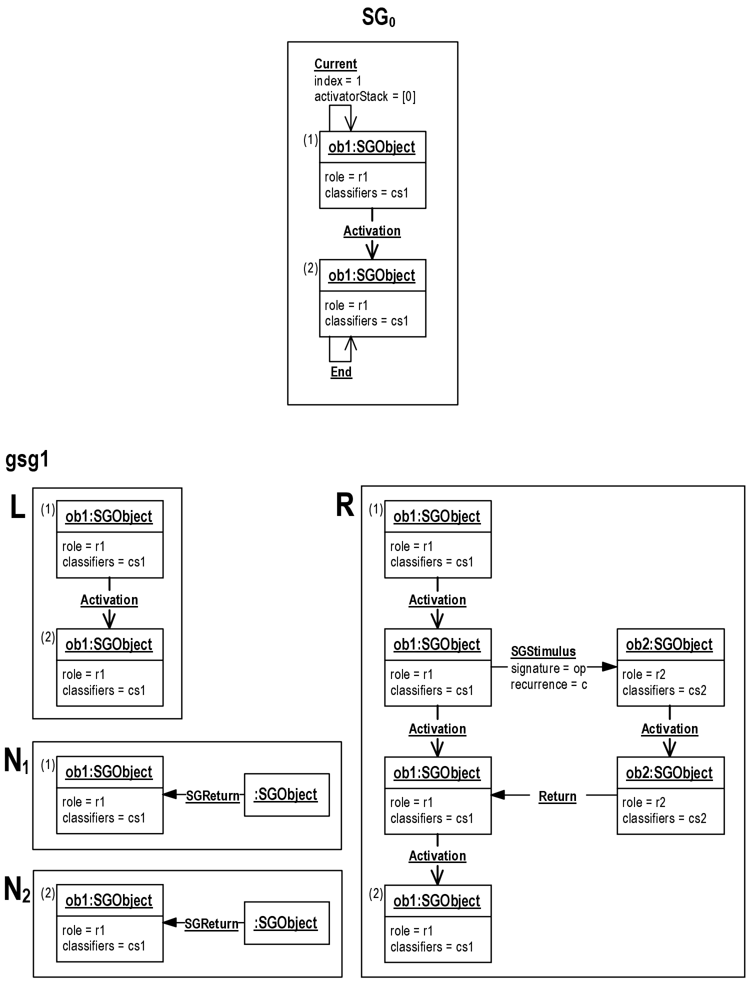

Fig. 5. Initial sequence graph and rule gsg1

is the source node of an SGStimulus edge and the second one is the target node of the matching Return edge. The attribute values of the SGStimulus edge can be freely chosen. The target node of the SGStimulus and the source node of the Return are added to the host graph as well. They form an activation path of their own. Their attribute values can be freely chosen and they might even equal those of the nodes of any other activation path. Note that the Instance they represent must possess the operation defined by the signature attribute of the SGStimulus edge. This rule enables one to model sequential and nested communication including recursive procedure calls of the same Instance.

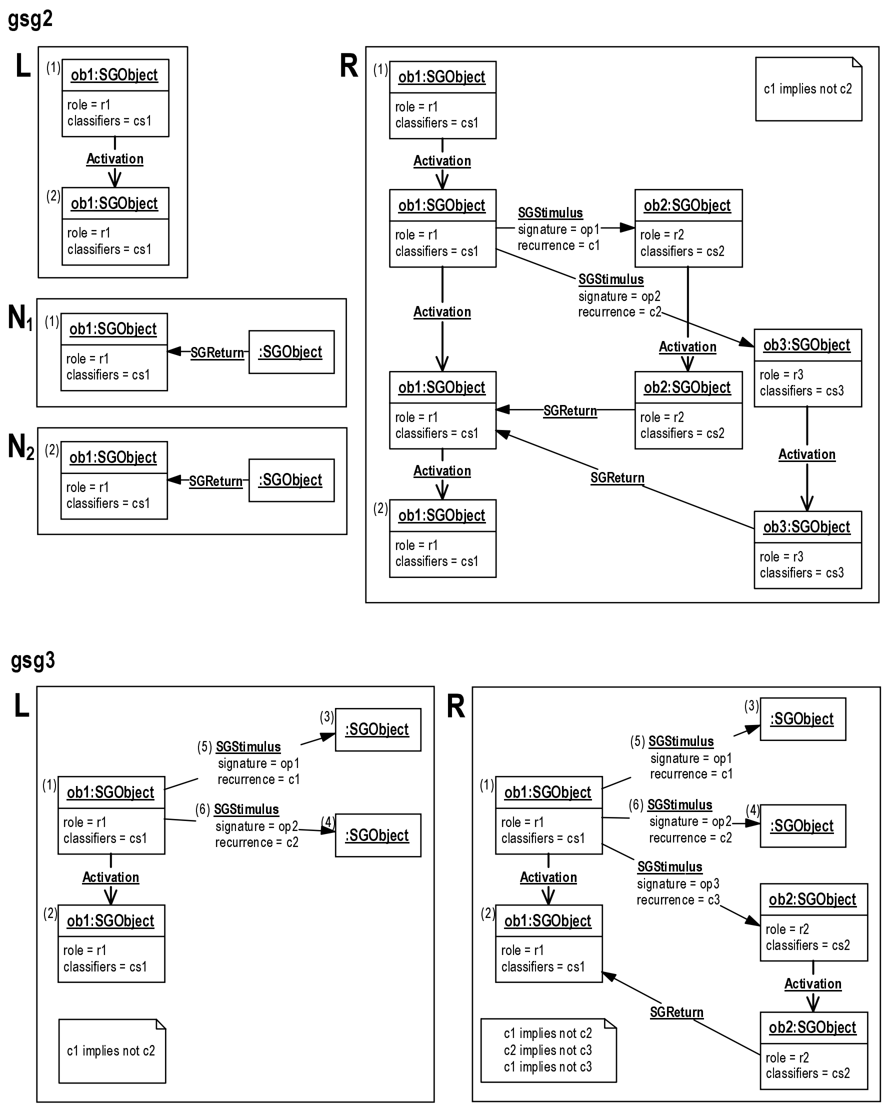

Fig. 6. Rules gsg2 and gsg3

Rule gsg2 is used to insert a conditional branching with two branches in an activation path. The left-hand side and the two negative application graphs N_1 and N_2 are identical to those of rule gsg1.

The right-hand side of this rule contains that of rule gsg1, i.e. the same changes are made. Additionally a second branch is inserted in the host graph starting at the source node of the newly added SGStimulus edge. It ends at the target node of the newly added Return. Note that both SGStimulus edges must have recurrence clauses and these must be mutually exclusive. This is necessary to ensure that the branching is not concurrent.

Rule gsg3 adds a new branch to an already existing conditional branching. Since an existing branching must always contain at least two branches and in order to avoid that a branching is created where none was before, the left-hand side must identify two different SGStimulus edges leaving the same source node. The end node of the conditional branching is also determined. It is the node directly succeeding the start node of the branching in the activation path. The application of the rule adds a new SGStimulus edge with a target node starting a new activation path to the host graph. A Return edge is also added which leaves the end of this activation path and points to the end node of the conditional branching. The recurrence clause of the newly added SGStimulus edge must be mutually exclusive with the ones of all the SGStimulus edges of the branching (which could in fact contain more than the ones identified by the left-hand side).

If a graph transformation unit is used as graph class expression, its semantics is the graph language consisting of all terminal graphs that are related to some initial graph. To make this notion feasible, we assume that there is only a single initial graph as in the case of **GenerateSG**. Formally, a graph is a graph class expression that specifies itself, i.e. $\mathrm{SEM}(G) = \{G\}$ for all graphs G. Note that **GenerateSG** has no import, the rules are not regulated and all graphs are terminal. Therefore the language consists of all graphs that can be derived from the initial graph by the three rules. This is the set of sequence graphs that serve as inputs of the translation into collaboration graphs. Note that these graphs resemble sequence diagrams, they are not meant as abstract syntax description.

5 Example

The short example presented in this section is taken from the model of a computer chess game. In the considered situation player Amy moves the white queen to a target field. If this is a diagonal move it is validated as the move of a bishop, otherwise as the move of a rook.

Figure 7 shows the sequence diagram of this situation, which contains nested communication and a conditional branching.

The collaboration part of the metamodel object graph generated by the application of **SG2MOG_Collaboration** is depicted in Figure 8.

Applying **SG2MOG_Interaction** yields the intermediate graph, which is omitted due to its complexity. Figure 9 shows only the activator tree, which is the most important part with respect to the translation.

Finally, Figure 10 shows the resulting collaboration graph after applying **MOG2CG** and the corresponding collaboration diagram.

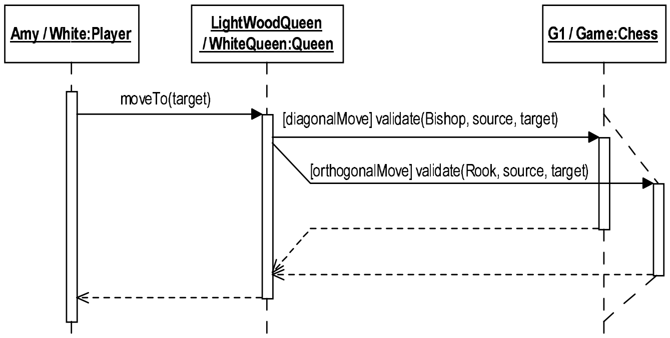
Amy / White:Player
LightWoodQueen / WhiteQueen:Queen
G1 / Game:Chess
moveTo(target)
[diagonalMove] validate(Bishop, source, target)
[orthogonalMove] validate(Rook, source, target)

Fig. 7. Sequence diagram

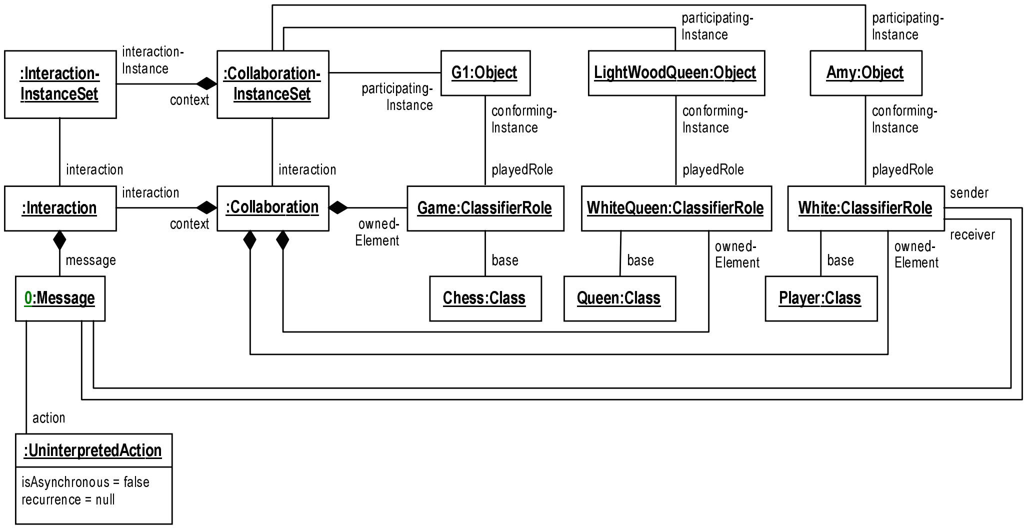

Fig. 8. Collaboration part of the metamodel object graph

6 Correctness

A translation can be called *correct* if the meaning of each input coincides with the meaning of the resulting output in a reasonable way. If both meanings are not equal, they should differ only in insignificant aspects (cf. [9]).

To consider the correctness of **SG2CG**, we must fix a meaning of sequential and collaboration diagrams. According to [2], the meaning of UML diagrams is given by their metamodel description besides the description in natural language. In other words, we may consider the translation **SG2MOG** of sequence graphs into metamodel object graphs (together with the one-to-one correspondence of these graphs with the respective types of diagrams) as the meaning of sequence

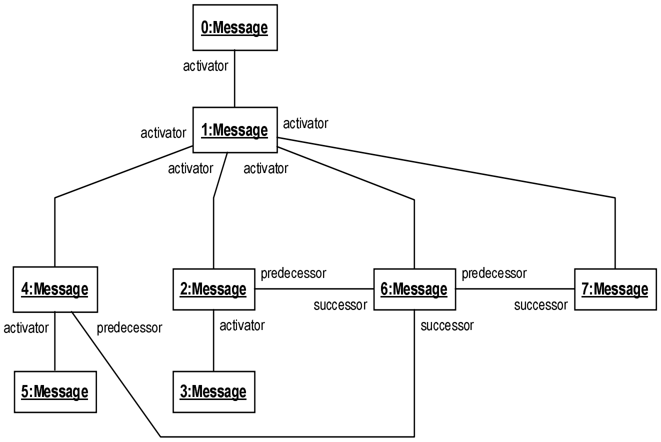

Fig. 9. Activator tree of the metamodel object graph

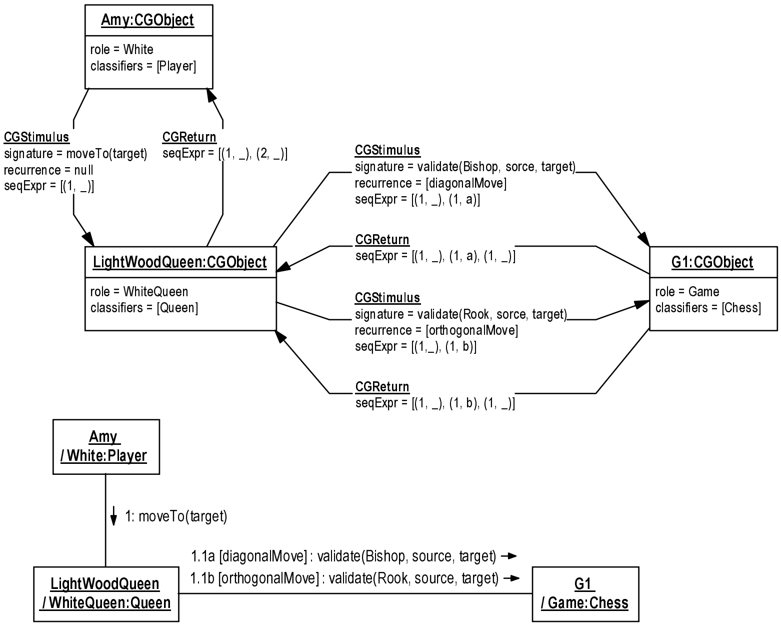
Amy:CGObject
role = White
classifiers = [Player]
CGStimulus
signature = moveTo(target)
recurrence = null
seqExpr = [(1, _)]
CGReturn
seqExpr = [(1, _), (2, _)]
CGStimulus
signature = validate(Bishop, sorce, target)
recurrence = [diagonalMove]
seqExpr = [(1, _), (1, a)]
CGReturn
seqExpr = [(1, _), (1, a), (1, _)]
LightWoodQueen:CGObject
role = WhiteQueen
classifiers = [Queen]
CGStimulus
signature = validate(Rook, sorce, target)
recurrence = [orthogonalMove]
seqExpr = [(1,_), (1, b)]
G1:CGObject
role = Game
classifiers = [Chess]
CGReturn
seqExpr = [(1, _), (1, b), (1, _)]
Amy
/ White:Player
1: moveTo(target)
1.1a [diagonalMove] : validate(Bishop, source, target)
1.1b [orthogonalMove] : validate(Rook, source, target)
LightWoodQueen
/ WhiteQueen:Queen
G1
/ Game:Chess

Fig. 10. Resulting collaboration graph and collaboration diagram

diagrams. Analogously, we consider a translation **CG2MOG** of collaboration diagrams onto the metamodel level to establish the meaning of collaboration diagrams.

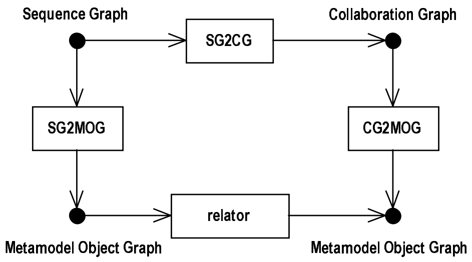

Fig. 11. Semantic relator discarding insignificant differences

This translation is given explicitly in the appendix of [3] and works analogously to translations presented in Section 3. As a consequence the correctness of our translation **SG2CG** requires to show that **SG2MOG** equals the composition of **SG2CG** and **CG2MOG** up to certain insignificant differences as illustrated in Figure 11.

Indeed, the metamodel object graph that is regarded as formal semantics of a sequence diagram is not necessarily unique. This is due to the fact, that in the case of conditional branchings the derivation process is not deterministic. The order in which the respective branches are translated is not fixed. So two translations of the same sequence graph may result in two different metamodel object graphs due to the numbering of the Messages during the translation process. Thus a different order in translating the branches of a conditional branching may yield different numbers for the involved Messages. This can be regarded as a minor aspect, since the decisive activator tree structure together with the predecessor-successor relationships is identical in both graphs. So in our case it makes sense to demand equality except for the numbering that is naming of the Messages. The numbering of the Messages has been established by us to aid in the translation. They are not demanded by the UML metamodel and must be omitted in the comparison of the metamodel object graphs. This can be realized by introducing a further transformation unit used as a semantic relator (cf. [9]), the purpose of which is to make the compared objects semantically equivalent by removing the insignificant differences. After applying this semantic relator to the regarded metamodel object graphs they must be equal.

This can be proved by complete induction on the lengths of derivations of the transformation unit **GenerateSG** that start with the initial sequence graph because, according to the construction in Section 4, the graphs derived in this way are the inputs of the translation.

The induction basis is a derivation length of zero. In this case, the transformation unit **GenerateSG** yields its initial graph as depicted in Figure 5. Now all the transformation units are applied, yielding the corresponding collaboration graph. This collaboration graph and the original sequence graph are then translated to their respective metamodel object graph by means of **CG2MOG** and **SG2MOG**. In this case the two resulting metamodel object graphs are already equal, since they both contain only one Message numbered 0. So the basis of the induction is proved.

The inductive step is set up by the statement, that we assume a correct translation for a derivation length of n and that we want to prove the correctness for a derivation length of $n + 1$. Since the transformation unit **GenerateSG** has three local rules, there are three cases that have to be distinguished in the inductive step. Each of these cases then has a number of subcases, depending on the concrete application situation. For example rule gsg1 can be used to either add a procedure call to a sequential communication or to insert it into a nested communication. For every case it has to be checked how the metamodel object graphs have to be changed during the translation of the generated sequence graphs in $n + 1$ derivation steps compared to the one generated in the first n identical derivation steps. So basically it suffices to compare the changes in the metamodel object graphs for every case after applying the semantic relators. This would prove that our approach yields a correct translation. But the explicit and detailed presentation of the various cases is beyond the scope of this paper.

7 Conclusion

We have presented a translation of UML sequence diagrams into UML collaboration diagrams in a structured way and sketched how the correctness can be proved. Due to Booch, Jacobson and Rumbaugh [2], the semantics of both types of diagrams is given in terms of metamodel object diagrams such that the correctness proof must involve this intermediate level.

We have constructed the translator in the framework of graph transformation units (cf. [10],[11]) because their interleaving semantics establishes translations between initial and terminal graphs by definition, supports correctness proofs by an underlying induction schema, and provides structuring concepts.

The translation of sequence diagrams into collaboration diagrams may also be seen as an example of the more recently introduced notion of a model transformation (see, e.g., [1], [4], [19]). But these approaches do not yet seem to support structuring and correctness proofs explicitly in the way they are used in this paper.

References

[1] D. Akehurst and S. Kent. A Relational Approach to Defining Transformations in a Metamodel. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *Proc. 5th Int. Conference on UML 2002—The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2002. 289

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998. 275, 286, 289

[3] B. Cordes and K. Hölscher. UML Interaction Diagrams: Correct Translation of Sequence Diagrams into Collaboration Diagrams. Diploma thesis, Department of Computer Science, University of Bremen, Bremen, Germany, 2003. 278, 288

[4] J. de Lara and H. Vangheluwe. AToM^3: A Tool for Multi-formalism and Meta-modelling. In R.-D. Kutsche and H. Weber, editors, *Proc. 5th Int. Conference on Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 2002. 289

[5] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach. In G. Rozenberg, editor, *The Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997. 277, 278

[6] M. Gogolla. Graph Transformations on the UML Metamodel. In J. D. P. Rolim, A. Z. Broder, A. Corradini, R. Gorrieri, R. Heckel, J. Hromkovic, U. Vaccaro, and J. B. Wells, editors, *Proc. ICALP Workshop Graph Transformations and Visual Modeling Techniques (GVMT'2000)*, pages 359–371. Carleton Scientific, Waterloo, Ontario, Canada, 2000. 276

[7] M. Gogolla and F. Parisi-Presicce. State Diagrams in UML: A Formal Semantics using Graph Transformations. In M. Broy, D. Coleman, T. S. E. Maibaum, and B. Rumpe, editors, *Proc. ICSE'98 Workshop on Precise Semantics for Modeling Techniques*, pages 55–72. Technical Report TUM-I9803, 1998. 276

[8] M. Gogolla, P. Ziemann, and S. Kuske. Towards an Integrated Graph Based Semantics for UML. In P. Bottoni and M. Minas, editors, *Proc. ICGT Workshop Graph Transformation and Visual Modeling Techniques (GT-VMT'2002)*, volume 72(3) of *Electronic Notes in Theoretical Computer Science*. Springer, 2002. 276

[9] H.-J. Kreowski. Translations into the Graph Grammar Machine. In R. Sleep, R. Plasmeijer, and M. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 171–183. John Wiley, New York, 1993. 286, 288

[10] H.-J. Kreowski and S. Kuske. On the Interleaving Semantics of Transformation Units—A Step into GRACE. In J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*, pages 89–106. Springer, 1996. 278, 289

[11] H.-J. Kreowski and S. Kuske. Graph Transformation Units and Modules. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *The Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999. 278, 289

[12] S. Kuske. A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 241–256. Springer, 2001. 276

[13] S. Kuske, M. Gogolla, R. Kollmann, and H.-J. Kreowski. An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In M. Butler and K. Sere, editors, *3rd Int. Conf. Integrated Formal Methods (IFM'02)*, volume 2335 of *Lecture Notes in Computer Science*. Springer, 2002. 276

[14] M. Löwe, M. Korff, and A. Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In R. Sleep, R. Plasmeijer, and M. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 185–199. John Wiley, New York, 1993. 277

[15] A. Maggiolo-Schettini and A. Peron. Semantics of Full Statecharts Based on Graph Rewriting. In H.-J. Schneider and H. Ehrig, editors, *Proc. Graph Transformation in Computer Science*, volume 776 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 1994. 276

[16] A. Maggiolo-Schettini and A. Peron. A Graph Rewriting Framework for Statecharts Semantics. In J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 1996. 276

[17] OMG, editor. OMG Unified Modeling Language Specification, Version 1.4, September 2001. Technical report, Object Management Group, Inc., Framingham, MA, 2001. 275

[18] A. Tsiolakis and H. Ehrig. Consistency Analysis of UML Class and Sequence Diagrams using Attributed Graph Grammars. In H. Ehrig and G. Taentzer, editors, *Proc. of Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems*, pages 77–86, 2000. Technical Report no. 2000/2, Technical University of Berlin. 276

[19] D. Varró. A Formal Semantics of UML Statecharts by Model Transition Systems. In A. Corradini, H. Ehrig, H. J. Kreowski, and G. Rozenberg, editors, *Proc. First Int. Conference on Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 378–392. Springer, 2002. 289