Strengthening UML Collaboration Diagrams by State Transformations*

Reiko Heckel and Stefan Sauer

University of Paderborn, Dept. of Mathematics and Computer Science
D-33095 Paderborn, Germany
reiko|sauer@uni-paderborn.de

Abstract. Collaboration diagrams as described in the official UML documents specify patterns of system structure and interaction. In this paper, we propose their use for specifying, in addition, pre/postconditions and state transformations of operations and scenarios. This conceptual idea is formalized by means of graph transformation systems and graph process, thereby integrating the state transformation with the structural and the interaction aspect.

Keywords: UML collaboration diagrams, pre/postconditions, graph transformation, graph process

1 Introduction

The Unified Modeling Language (UML) [24] provides a collection of loosely coupled diagram languages for specifying models of software systems on all levels of abstraction, ranging from high-level requirement specifications over analysis and design models to visual programs. On each level, several kinds of diagrams are available to specify different aspects of the system, like the structural, functional, or interaction aspect. But even diagrams of the same kind may have different interpretations when used on different levels, while several aspects of the same level may be expressed within a single diagram.

For example, interaction diagrams in UML, like sequence or collaboration diagrams, often represent sample communication scenarios, e.g., as refinement of a use case, or they may be used in order to give a complete specification of the protocol which governs the communication. Collaboration diagrams allow, in addition, to represent individual snapshots of the system as well as structural patterns.

If such multi-purpose diagrammatic notations shall be employed successfully, a precise understanding of their different aspects and abstraction levels is required, as well as a careful analysis of their mutual relations. This understanding, once suitably formalized, can be the basis for tool support of process models, e.g., in the form of consistency checks or refinement rules.

* Research partially supported by the ESPRIT Working Group APPLIGRAPH.

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 109–123, 2001.

In this paper, we address these issues for UML collaboration diagrams. These diagrams are used on two different levels, the instance and the specification level, both related to a class diagram for typing (cf. Fig. 1). A specification-level diagram provides a pattern which may occur at the instance level.[1]

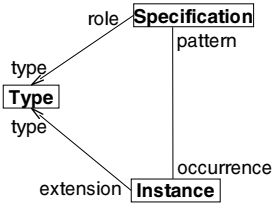

Fig. 1. Two levels of collaboration diagrams and their typing

In addition, in the UML specification [24] two aspects of collaboration diagrams are identified: the *structural aspect* given by the graph of the collaboration, and the *interaction aspect* represented by the flow of messages. These aspects are orthogonal to the dimensions in Fig. 1: A specification-level diagram may provide a structural pattern as well as a pattern of interaction. At the instance level, a collaboration diagram may represent a snapshot of the system or a sample interaction scenario. Moreover, both aspects are typed over the class diagram, and the pattern-occurrence relation should respect this typing.

One way to make precise the relationships between different diagrams and abstraction levels is the approach of *meta modeling* used in the UML specification [24]. It allows to specify the syntactic relation between different diagrams (or different uses of the same diagram) by representing the entire model by a single *abstract syntax graph* where dependencies between different diagrams can be expressed by means of additional links, subject to structural constraints specifying consistency. This approach provides a convenient and powerful language for integrating diagram languages, i.e., it contributes to the question, *how* the integration can be specified. However, it provides no guidelines, *what* the intended relationships between different diagrams should be.

Therefore, in this paper, we take the alternative approach of translating the diagrams of interest into a formal method which is conceptually close enough in order to provide us with the required semantic intuition to answer the *what* question. Once this is sufficiently understood, the next step is to formulate these results in the language of the UML meta model.

Our formal method of choice are graph transformation systems of the so-called *algebraic double-pushout (*DPO*) approach* [10] (see [5] for a recent survey).

[1] The use of collaboration diagrams for *role modeling* is not captured by this picture. A role model provides a refinement of a class diagram where roles restrict the features of classes to those relevant to a particular interaction. A collaboration diagram representing a role model can be seen as a second level of typing for instance (and specification-level) diagrams which itself is typed over the class diagram. For simplicity, herein we restrict ourselves to a single level of typing.

In particular, their typed variant [4] has built in most of the aspects discussed above, including the distinction between pattern, instance, and typing, the structural aspect and (by way of the partial order semantics of *graph processes* [4]) a truly concurrent model for the interaction aspect. The latter is in line with the recent proposal for UML action semantics [1] which identifies a semantic domain for the UML based on a concurrent data flow model.

The direct interpretation of class and collaboration diagrams as graphs and of their interrelations as graph homomorphisms limits somewhat the scope of the formalization. In particular, we deliberately neglect inheritance, ordered or qualified associations, aggregation, and composition in class diagrams as well as multi-objects in collaboration diagrams. This oversimplification for presentation purpose does not imply a general limitation of the approach as we could easily extend the graph model in order to accommodate these features, e.g., using a meta model-based approach like in [21].

Along with the semantic intuition gained through the interpretation of collaboration diagrams in terms of graph transformation comes a conceptual improvement: the use of collaboration diagrams as a visual query and update language for object structures. In fact, in addition to system structure and interaction, we propose the use of collaboration diagrams for specifying the *state transformation* aspect of the system. So far, this aspect has been largely neglected in the standard documents [24], although collaboration diagrams are used already in the CATALYSIS approach [6] and the FUSION method [3] for describing pre- and postconditions of operations and scenarios.

Beside a variety of theoretical studies, in particular in the area of concurrency and distributed systems [9], application-oriented graph transformation approaches like PROGRES [29] or FUJABA [14] provide a rich background in using rule-based graph transformation for system modeling as well as for testing, code generation, and rapid prototyping of models (see [7] for a collection of survey articles on this subject). Recently, graph transformations have been applied to UML meta modeling, e.g., in [16,2,11].

Therefore, we believe that our approach not only provides a clarification, but also a conceptual improvement of the basic concepts of collaboration diagrams.

Two approaches which share the overall motivation of this work remain to be discussed, although we do not formally relate them herein. Övergaard [26] uses sequences in order to describe the semantics of interactions, including notions of refinement and the relation with use cases. The semantic intuition comes from trace-based interleaving models which are popular, e.g., in process algebra. Knapp [22] provides a formalization of interactions using temporal logic and the *pomset* (partially ordered multi-set) model of concurrency [27]. In particular, the pomset model provides a semantic framework which has much similarity with graph processes, only that pomsets are essentially set-based while graph processes are about graphs, i.e., the structural aspect is already built in. Besides technical and methodological differences with the cited approaches, the main additional objective of this work is to *strengthen collaboration diagrams by incorporating graph transformation concepts.*

The presentation is structured according to the three aspects of collaboration diagrams. After introducing the basic concepts and a running example in Sect. 2, Sect. 3 deals with the structural and the transformation aspect, while Sect. 4 is concerned with interactions. Section 5 concludes the paper.

A preliminary sketch of the ideas of this paper has been presented in [20].

2 UML Collaboration Diagrams: A Motivating Example

In this section, we introduce a running example to motivate and illustrate the concepts in this paper. First, we sketch the use of collaboration diagrams as suggested in the UML specification [24]. Then, we present an improved version of the same example exploiting the state transformation aspect.

Figure 2 shows the class diagram of a sample application where a Company object is related to zero or more Store, Order, and Delivery objects. Order objects as well as Delivery objects are related to exactly one Customer who plays the role of the customer placing the order or the receiver of a delivery, respectively. A Customer can place several instances of Order and receive an unrestricted number of Delivery objects.

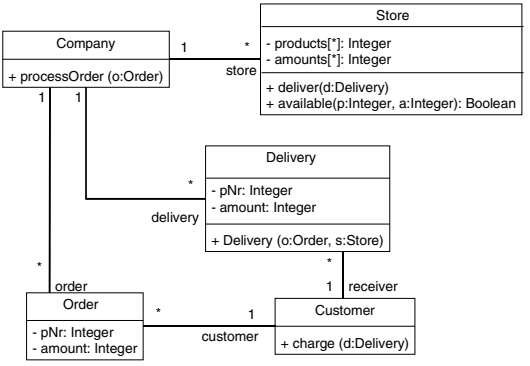

Fig. 2. A class diagram defining the structure of the example

A typical scenario within that setting is the situation where a customer orders a product from the company. After the step of refining and combining different use cases into a method-oriented specification one might end up with a collaboration diagram specifying the implementation of operation processOrder as depicted in the top of Fig. 3. Here, the company first obtains the product number pNr and the ordered amount using defined access functions. It then checks all stores to find one that can supply the requested amount of the demanded product. A delivery is created, and the selected store is called to send it out. Concurrently, the customer is charged for this delivery. After an order has been processed, it will be deleted.

Collaboration diagrams like this, which specifies the execution of an operation, may be used for generating method implementations in Java [12], i.e., they

can be seen as visual representations of programs. However, in earlier phases of development, a higher-level style of specification is desirable which abstracts from implementation details like the get functions for accessing attributes and the implementation of queries by search functions on multi-objects.

Therefore, we propose to interpret a collaboration as a visual query which uses pattern matching on objects, links, and attributes instead of low-level access and search operations. In fact, leaving out these details, the same operation can be specified more abstractly by the diagram in the lower left of Fig. 3. Here, the calls to getpNr and getAmount are replaced by variables p and a for the corresponding attribute values, and the call of search on the multi-object is replaced by a boolean function available which constrains the instantiation of /s:Store. (As specified in the lower right of the same figure, the function returns true if the Store object matching /s is connected to a Product object with the required product number p and an amount b greater than a.) The match is complete if all items in the diagram not marked as {new} are instantiated. Then, objects marked as {destroyed} are removed from the current state while objects marked as {new} are created, initializing appropriately the attributes and links. For example, the new Delivery object inherits its link and attribute values from the destroyed Order object.

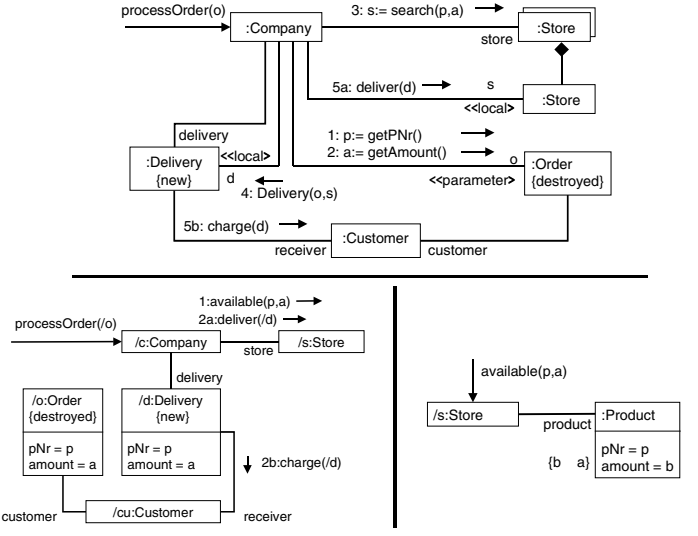

Fig. 3. An implementation-oriented collaboration diagram (top), its declarative presentation (bottom left), and a visual query operation (bottom right)

In the following sections, we show how this more abstract use of collaboration diagrams can be formalized by means of graph transformation rules and graph processes.

3 Collaborations as Graph Transformations

A collaboration on specification level is a graph of classifier roles and association roles which specifies a view of the classes and associations of a class diagram as well as a pattern for objects and links on the instance level. This triangular relationship, which instantiates the type-specification-instance pattern of Fig. 1 for the structural aspect, shall be formalized in the first part of this section. Then, the state transformation aspect shall be described by means of graph transformations. The interaction aspect is considered in the next section.

Structure. Focusing on the structural aspect first, we use graphs and graph homomorphisms (i.e., structure-compatible mappings between graphs) to describe the interrelations between class diagrams and collaboration diagrams on the specification and the instance level.

The relation between class and instance diagrams is formally captured by the concept of *type* and *instance graphs* [4]. By *graphs* we mean directed unlabeled graphs $G = \langle G_V, G_E, src^G, tar^G \rangle$ with set of vertices G_V, set of edges G_E, and functions $src^G : G_E \to G_V$ and $tar^G : G_E \to G_V$ associating to each edge its source and target vertex, respectively. A graph homomorphism $f : G \to H$ is a pair of functions $\langle f_V : G_V \to H_V, f_E : G_E \to H_E \rangle$ compatible with source and target, i.e., for all edges e in G_E, $f_V(src^G(e)) = src^H(f_E(e))$ and $f_V(tar^G(e)) = tar^H(f_E(e))$.

Let TG be the underlying graph of a class diagram, called *type graph*. A legal *instance graph* over TG consists of a graph G together with a typing homomorphism $g : G \to TG$ associating to each vertex and edge x of G its type $g(x) = t$ in TG. In UML notation, we write $x : t$. Observe that the compatibility of g with source and target ensures that, e.g., the class of the source object of a link is the source class of the link's association. Constraints like this can be found in the meta class diagrams and well-formedness rules of the UML meta model for the meta associations relating classifiers with instances, associations with links, association ends with link ends, etc. ([24], Sect. 2.9). The typing of specification-level graphs is described in a similar way ([24], Sect. 2.10).

An interpretation of a graph homomorphism which is conceptually different, but requires the same notion of structural compatibility, is the occurrence of a pattern in a graph. For example, a collaboration on the specification level occurs in a collaboration on the instance level if there exists a mapping from classifier roles to instances and from association roles to links preserving the connections. Thus, the existence of a graph homomorphism from a given pattern graph implies the presence of a corresponding instance level structure. The occurrence has to be type-compatible, i.e., if a classifier role is mapped to an instance, both have to be of the same classifier. This compatibility is captured in the notion of a *typed graph homomorphism* between typed graphs, i.e., a graph homomorphism which preserves the typing. In our collaboration diagrams, this concept of graphical pattern matching is used to express visual queries on object structures.

In summary, class and collaboration diagrams have a homogeneous, graph-like structure, and their triangular relationship can be expressed by three compatible graph homomorphisms. Next, this triangular relation shall be lifted to the state transformation view.

State Transformation. Collaborations specifying queries and updates of object structures are formalized as *graph transformation rules*, while corresponding collaborations on the instance level represent individual *graph transformations*.

A *graph transformation rule* $r = L \to R$ consists of two graphs L, R such that the union $L \cup R$ is defined. (This ensures that, e.g., edges which appear in both L and R are connected to the same vertices in both graphs.) Consider the rule in the upper part of Fig. 4 representing the collaboration of processOrder in the lower left of Fig. 3. The precondition L contains all objects and links which have to be present before the operation, i.e., all elements of the diagram except for /d:Delivery which is marked as {new}. Analogously, the postcondition R contains all elements except for /o:Order which is marked as {destroyed}. (The {transient} constraint does not occur because a graph transformation rule is supposed to be atomic, i.e., conceptually there are no intermediate states between L and R.)

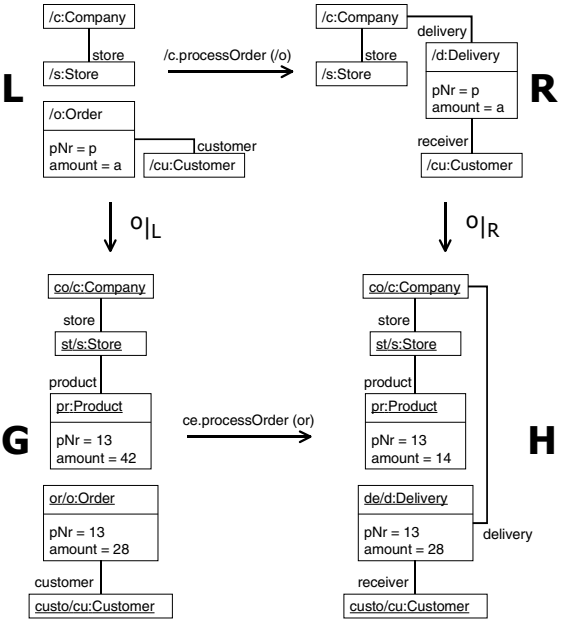

Fig. 4. A graph transition consisting of a rule $L \to R$ specifying the operation processOrder (top), and its occurrence o in an instance-level transformation (bottom)

A similar diagram on the instance level represents a graph transformation. Graph transformation rules can be used to specify transformations in two different ways: either operationally by requiring that the rule is applied to a given graph in order to rewrite part of it, or axiomatically by specifying pre- and postconditions. In the first interpretation (in fact, the classical one [10], in set-theoretic formulation), a *graph transformation* $G \overset{r(o)}{\Longrightarrow} H$ from a pre-state G to a post-state H using rule r is represented by a graph homomorphism $o : L \cup R \to G \cup H$, called *occurrence*, such that

1. $o(L) \subseteq G$ and $o(R) \subseteq H$ (i.e., the left-hand side of the rule is matched by the pre-state and the right-hand side by the post-state),
2. $o(L \setminus R) = G \setminus H$ and $o(R \setminus L) = H \setminus G$ (i.e., all objects of G are {destroyed} that match classifier roles of L not belonging to R and, symmetrically, all objects of H are {new} that match classifier roles in R not belonging to L).

That is, the transformation creates and destroys exactly what is specified by the rule and the occurrence. As a consequence, the rule together with the occurrence of the left-hand side L in the given graph G determines, up to renaming, the derived graph H, i.e., the approach has a clear operational interpretation, which is well-suited for visual programming.

In the more liberal, axiomatic interpretation, requirement 2 is weakened to

2′. $o(L \setminus R) \subseteq G \setminus H$ and $o(R \setminus L) \subseteq H \setminus G$ (i.e., *at least* the objects of G are {destroyed} that match classifier roles of L not belonging to R and, symmetrically, *at least* the objects of H are {new} that match classifier roles in R not belonging to L).

These so-called *graph transitions* [19] allow side effects not specified by the rule, like in the example of Fig. 4 where the amount of product pr changes without being explicitly rewritten by the rule. This is important for high-level modeling where specifications of behavior are often incomplete.

In both cases, instance transformations as well as specification-level rules are typed over the same type graph, and the occurrence homomorphism respects these types. This completes the instantiation of the type-specification-instance pattern for the aspect of state transformation.

Summarizing, a collaboration on the specification level represents a pattern for state transformations on the instance level, and the occurrence of this pattern requires, beside the structural match of the pre- and postconditions, (at least) the realization of the described effects. Graph transformations provide a formal model for the state transformation aspect which allows to describe the overall effect of a complex interaction. However, the interaction itself, which decomposes the global steps into more basic actions, is not modeled. In the next section, this finer structure shall be described in terms of the model of concurrency for graph transformation systems [4].

4 Interactions as Graph Processes

In this section, we shall extend the triangular type-specification-instance pattern to the interaction part. First, we describe the formalization of the individual concepts and then the typing and occurrence relations.

Class diagrams. A class diagram is represented as a *graph transformation system*, briefly GTS, $\mathcal{G} = \langle TG, \mathcal{R} \rangle$ consisting of a type graph TG and a set of transformation rules $\mathcal{R}$. The type graph captures the structural aspect of the class diagram, like the classes, associations, and attributes, as well as the types of call and return messages that are sent when an operation is invoked. For the

fragment of the class diagram consisting of the classes Order, Customer, and Delivery, the customer association[2], and the attributes and operations of the first two classes, the type graph is shown in Fig. 5. Classes are as usually shown as rectangular, data types as oval shapes. Call and return messages are depicted like UML action states, i.e., nodes with convex borders at the two sides. Return messages are marked by overlined labels. Links from call message nodes represent the input parameters of the operation, while links from return message nodes point to output parameters (if any). The self link points to the object executing the operation. The rules of the GTS in Fig. 5 model different kinds of basic actions that are implicitly declared within the class diagram. Among them are state transformation actions like destroy o, control actions like send charge, and actions representing the execution of an operation like cu.charge(d).

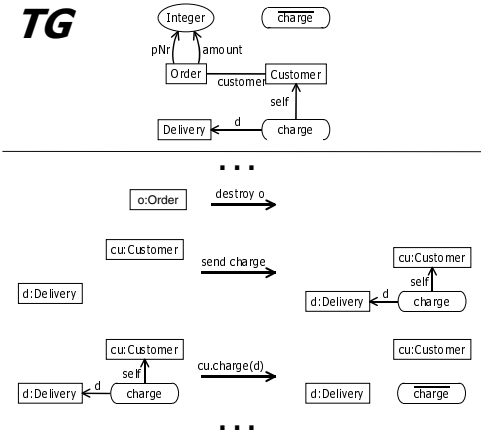

Fig. 5. Graph transformation system for a fragment of the class diagram in Fig. 2

Interactions. An interaction consists of a set of messages, linked by control and data flow, that stimulate actions like access to attributes, invocation of operations, creation and deletion of objects, etc. While the control flow is explicitly given by sequence numbers specifying a partial order over messages, data flow information is only implicitly present, e.g., in the parameters of operations and in their implementation, as far as it is given. However, it is important that control and data flow are compatible, i.e., they must not create cyclic dependencies. Such constraints are captured by the concept of a *graph process* which provides a partial order semantics for graph transformation systems.

The general idea of process semantics, which have their origin in the theory of Petri nets [28], is to abstract, in an individual run, from the ordering of actions that are not causally dependent, i.e., which appear in this order only by accident or because of the strategy of a particular scheduler. If actions are represented

[2] More precisely, in UML terms this is an unnamed association in which class Customer plays the role customer.

by graph transformation rules specifying their behavior in a pre/postcondition style, these causal dependencies can be derived by analyzing the intersections of rules in the common context provided by the overall collaboration.

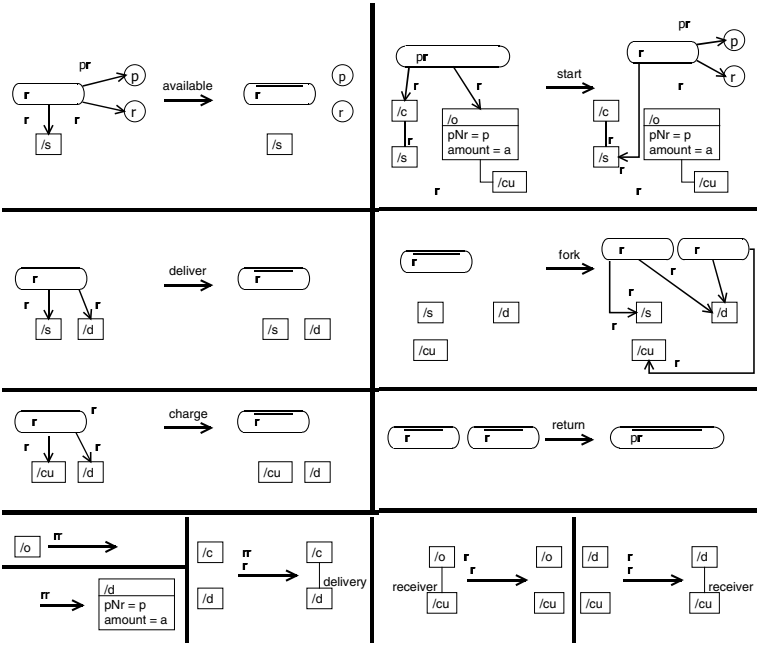

Fig. 6. Graph process for the collaboration diagram of operation processOrder. The three rules in the upper left section represent the operations, those in the upper right realize the control flow between these operations, and the five rules in the lower part are responsible for state transformations (note that attribute values a and p of /d can be instantiated by new /d since all the rules of the graph process act in a common context given by the collaboration)

The graph process for the collaboration diagram in the lower left of Fig. 3 is shown in Fig. 6. It consists of a set of rules representing the internal actions, placed in a common name space. That means, e.g., the available node created by the rule start in the top right is the same as the one deleted by the rule available in the top left. Because of this causal dependency, available has to be performed after start. Thus, the causality of actions in a process is represented by the overlapping of the left- and right-hand sides of the rules.

Graph processes are formally defined in three steps. A *safe graph transformation system* consists of a graph C (best to be thought of as the graph of the collaboration) together with a set of rules $\mathcal{T}$ such that, for every rule $t = G \to H \in \mathcal{T}$ we have $G, H \subseteq C$ (that is, C provides a common context for the rules in $\mathcal{T}$). Intuitively, the rules in $\mathcal{T}$ represent transformations, i.e., occurrences of rules. In order to formalize this intuition, the notion of *occurrence graph transformation system* is introduced requiring, in addition, that the

transformations in $\mathcal{T}$ can be ordered in a sequence. That means, the system has to be acyclic and free of conflicts, and the causality relation has to be compatible with the graphical structure. In order to make this precise, we define the causal relation associated to a safe GTS $\langle C, \mathcal{T} \rangle$. Let $t : G \to H$ be one arbitrary transformation in $\mathcal{T}$ and e be any edge, node, or attribute in C. We say that

- t *consumes* e if $e \in G \setminus H$
- t *creates* e if $e \in H \setminus G$
- t *preserves* e if $e \in G \cap H$

The relation $\leq$ is defined on $\mathcal{T} \cup C$, i.e., it relates both graphical elements and operations. It is the transitive and reflexive closure of the relation $<$ where

- $e < t_1$ if t_1 consumes e
- $t_1 < e$ if t_1 creates e
- $t_1 < t_2$ if t_1 creates e and t_2 preserves e, or
  t_1 preserves e and t_2 consumes e

Now, a safe GTS is called an *occurrence graph transformation system* if

- the causal relation $\leq$ is a partial order which respects source and target, i.e., if $t \in \mathcal{T}$ is a rule and a vertex v of C is source (or target) of an edge e, then
  - $t \leq v$ implies $t \leq e$ and
  - $v \leq t$ implies $e \leq t$
- for all elements x of C, x is consumed by at most one rule in $\mathcal{T}$, and it is created by at most one rule in $\mathcal{T}$.

The objective behind these conditions is to ensure that each occurrence GTS represents an equivalence class of sequences of transformations "up-to-rescheduling", which can be reconstructed as the linearizations of the partial order $\leq$. Vice versa, from each transformation sequence one can build an occurrence GTS by taking as context C the colimit (sum) of all instance graphs in the sequence [4].

The causal relation between the rules in the occurrence GTS in Fig. 6 is visualized by the Petri net in the left of Fig. 7. For example, the dependency between available and start discussed above is represented by the place between the corresponding transitions. Of these dependencies, which include both control- and data-flow, in the UML semantics only the control-flow dependencies are captured by a precedence relation on messages, specified using sequence numbers. This part is presented by the sub-net in the upper right of Fig. 7. In comparison, the net in the lower right of Fig. 7 visualizes the control flow if we replace the synchronous call to deliver by an asynchronous one: The call is delegated to a thread which consumes the return message and terminates afterwards. This strategy for modeling asynchronous calls might seem a little *ad-hoc*, but it follows the implementation of asynchronous calls in Java as well as the formalization of asynchronous message passing in process calculi [23]. The essential property is the independence of the deliver and the charge action.

We have used graph transformation rules for specifying both the overall effect of an interaction as well as its basic, internal actions. A fundamental fact about

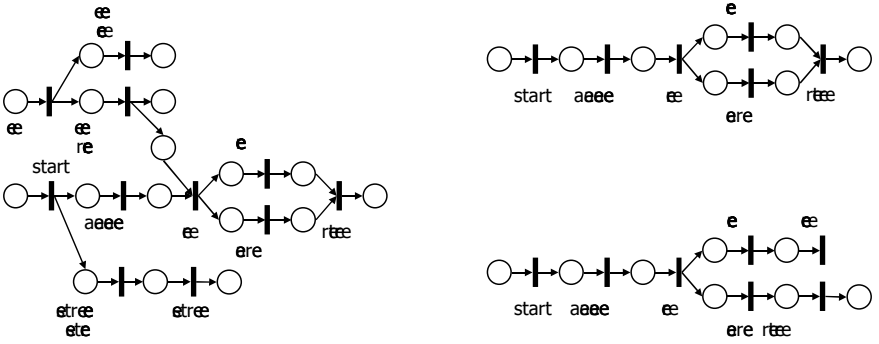

Fig. 7. Control flow and data dependencies of the occurrence GTS in Fig. 6 (left), sub-net for control flow dependencies of occurrence GTS (top right), and control flow dependencies with asynchronous call of deliver (bottom right)

occurrence GTS [4] relates the state transformation with the interaction aspect: Given an occurrence GTS $\mathcal{O} = \langle C, \mathcal{T} \rangle$ and its partial order $\leq$, the sets of minimal and maximal elements of C w.r.t. $\leq$ form two graphs $Min(\mathcal{O}), Max(\mathcal{O}) \subseteq C$. This allows us to view a process p externally as a transformation rule $\tau(p)$, called *total rule of* p, which combines the effects of all the local rules of $\mathcal{T}$ in a single, atomic step. The total rule of the process in Fig. 6 is shown in the top of Fig. 4.

Summarizing, the three corners of our triangle are represented by a GTS representing the type level, and two occurrence GTS formalizing interactions on the specification and the instance level, respectively. It remains to define the relation between these three levels, i.e., the concepts of typing and occurrence.

Typing. In analogy with the typing of graphs, an occurrence GTS $\mathcal{O} = \langle C, \mathcal{T} \rangle$ is typed over a GTS $\mathcal{G} = \langle TG, \mathcal{R} \rangle$ via a *homomorphism of graph transformation systems*, briefly GTS morphism. A GTS morphism $p : \mathcal{O} \to \mathcal{G}$ consists of a graph homomorphism $c : C \to TG$ typing the context graph C over the type graph TG, and a mapping of rules $f : \mathcal{T} \to \mathcal{R}$ such that, for every $t \in \mathcal{T}$, the rules t and $f(t)$ are equal up to renaming. Such a *typed* occurrence GTS is called a *graph process* [4].

Since all graphs in the rules of Fig. 6 are well-typed over the type graph TG, their union C is typed over TG by the union of the typing homomorphisms of its subgraphs. The rules representing basic actions, like operation invocations and state transformations, can be mapped directly to rules in $\mathcal{R}$. The control flow rules, which are more complex, are mapped to compound rules derived from the elementary rules in $\mathcal{R}$.[3]

Occurrence. The occurrence of a specification-level interaction pattern at the instance level is described by a plain homomorphism of graph transformation

[3] Categorically, this typing of an occurrence GTS can be formalized as a Kleisli morphism mapping elementary rules to derived ones (see, e.g., [18,17] for similar approaches in graph transformation theory).

systems: We want the same granularity of actions on the specification and the instance level. An example of an instance-level collaboration diagram is given in Fig. 8. It does not contain additional actions (although this would be permitted

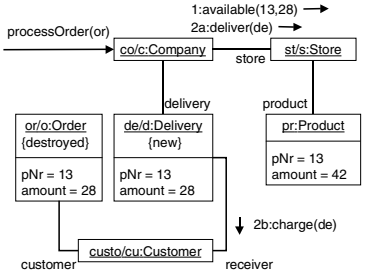

Fig. 8. Collaboration diagram on the instance level

by the definition of occurrence), but the additional context of the Product instance. In the process in Fig. 6, this would lead to a corresponding extension of the start rule.

As before, the GTS homomorphisms forming the three sides of the triangle have to be compatible. That means, an occurrence of a specification-level collaboration diagram in an instance-level one has to respect the typing of classes, associations, and attributes and of operations and basic actions.

This completes the formalization of the type-specification-instance triangle in the three views of collaboration diagrams of structure, state transformation, and interaction.

5 Conclusion

In this paper, we have proposed a semantics for collaboration diagrams based on concepts from the theory of graph transformation. We have identified and formalized three different aspects of a system model that can be expressed by collaboration diagrams (i.e., structure, state transformation, and interaction) and, orthogonally, three levels of abstraction (type, specification, and instance level). In particular, the idea of collaboration diagrams as state transformations provides new expressive power which has so far been neglected by the UML standard. The relationships between the different abstraction levels and aspects are described in terms of homomorphisms between graphs, rules, and graph transformation systems.

The next steps in this work consist in transferring the new insights to the UML specification. On the level of methodology and notation, the state transformation aspect should be discussed as one possible way of using collaboration diagrams. On the level of abstract syntax (i.e., the meta model) the pattern-occurrence relation between specification- and instance-level diagrams has to be made explicit, e.g., by additional meta associations. (In fact, this has been

partly accomplished in the most recent draft of the standard [25].) On the semantic level, a representation of the causal dependencies in a collaboration diagram is desirable which captures also the data flow between actions.

It remains to state more precisely the relation between collaboration diagrams as defined by the standard and our extended version. Obviously, although our collaboration diagrams are syntactically legal, the collaboration diagrams that are semantically meaningful according to the UML standard form a strict subset of our high-level diagrams based on graph matching. An implementation of this matching by explicit navigation (as it is given, for example, in [14] as part of a code generation in JAVA) provides a translation back to the original low-level style. The formal properties of this construction have not been investigate yet.

References

1. Action Semantics Consortium. Precise action semantics for the Unified Modeling Language, August 2000. `http://www.kc.com/as_site/`.
2. P. Bottoni, M. Koch, F. Parisi Presicce, and G. Taentzer. Consistency checking and visualization of OCL constraints. In Evans et al. [13], pages 294–308.
3. D. Coleman, P. Arnold, S. Bodof, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremes. *Object Oriented Development, The Fusion Method*. Prentice Hall, 1994.
4. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.
5. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, pages 163–245. World Scientific, 1997.
6. D. D'Souza and A. Wills. *Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.
7. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*. World Scientific, 1999.
8. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn, November 1998*, volume 1764 of *LNCS*. Springer-Verlag, 2000.
9. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency and Distribution*. World Scientific, 1999.
10. H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.
11. G. Engels, J.H. Hausmann, R. Heckel, and St. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In Evans et al. [13], pages 323–337.
12. G. Engels, R. Hücking, St. Sauer, and A. Wagner. UML collaboration diagrams and their transformation to Java. In France and Rumpe [15], pages 473–488.
13. A. Evans, S. Kent, and B. Selic, editors. *Proc. UML 2000 – Advancing the Standard*, volume 1939 of *LNCS*. Springer-Verlag, 2000.

14. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In Ehrig et al. [8].
15. R. France and B. Rumpe, editors. *Proc. UML'99 – Beyond the Standard*, volume 1723 of *LNCS*. Springer-Verlag, 1999.
16. M. Gogolla. Graph transformations on the UML metamodel. In J. D. P. Rolim et al., editors, *Proc. ICALP Workshops 2000, Geneva, Switzerland*, pages 359–371. Carleton Scientific, 2000.
17. M. Große-Rhode, F. Parisi Presicce, and M. Simeoni. Refinement of graph transformation systems via rule expressions. In Ehrig et al. [8], pages 368–382.
18. R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of typed graph transformation systems. *Math. Struc. in Comp. Science*, 6(6):613–648, 1996.
19. R. Heckel, H. Ehrig, U. Wolter, and A. Corradini. Double-pullback transitions and coalgebraic loose semantics for graph transformation systems. *Applied Categorical Structures*, 9(1), January 2001.
20. R. Heckel and St. Sauer. Strengthening the semantics of UML collaboration diagrams. In G. Reggio, A. Knapp, B. Rumpe, B. Selic, and R. Wieringa, editors, *UML'2000 Workshop on Dynamic Behavior in UML Models: Semantic Questions*, pages 63–69. October 2000. Tech. Report no. 0006, Ludwig-Maximilians-University Munich, Germany.
21. R. Heckel and A. Zündorf. How to specify a graph transformation approach: A meta model for FUJABA. In H. Ehrig and J. Padberg, editors, *Uniform Approaches to Graphical Process Specification Techniques, satellite workshop of ETAPS 2001, Genova, Italy*, 2001. To appear.
22. A. Knapp. A formal semantics of UML interactions. In France and Rumpe [15], pages 116–130.
23. M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *Proc. ICALP'98*, volume 1443 of *LNCS*, pages 856–867. Springer-Verlag, 1998.
24. Object Management Group. UML specification version 1.3, June 1999. `http://www.omg.org`.
25. Object Management Group. UML specification version 1.4beta R1, November 2000. `http://www.celigent.com/omg/umlrtf/`.
26. G. Övergaard. A formal approach to collaborations in the Unified Modeling Language. In France and Rumpe [15], pages 99–115.
27. V. Pratt. Modeling concurrency with partial orders. *Int. Journal. of Parallel Programming*, 15(1):33–71, February 1986.
28. W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.
29. A. Schürr, A.J. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Ehrig et al. [7], pages 487–550.