Coherently Explaining UML Statechart and Collaboration Diagrams by Graph Transformations

Paul Ziemann[1] **, Karsten Hölscher**[1] **, Martin Gogolla**[1]

[1]University of Bremen, Department of Computer Science
P.O. Box 330440, D-28334 Bremen, Germany

{ziemann|hoelsch|gogolla}@informatik.uni-bremen.de

Abstract. *In this paper we continue our work on the formalization and validation of UML models by means of graph transformation systems. We here concentrate on statechart and collaboration diagrams albeit our approach covers use case, class, object, and sequence diagrams as well. The statechart and collaboration diagrams describe the operations of the underlying class diagram and include OCL expressions as guards and parts of message expressions. We illustrate in detail the generation of graph transformation rules for the statechart and collaboration diagrams.*

1. Introduction

The Unified Modeling Language (UML) has recently become a widely accepted standard for the visualization, specification, construction, and documentation of object-oriented software systems. It is well established and used in industry as well as in research. The UML is a graphical language that comprises a number of different diagram types for different purposes. The syntax of these diagram types is defined in the UML metamodel [OMG, 2003]. But the semantics of the language constructs is only given in natural language. As the UML is supposed to support a software engineer in constructing precise models, a formal foundation for UML is needed. The graphical notation is enhanced by the Object Constraint Language (OCL), which permits to formulate constraints that cannot be expressed by the diagrams in a textual way. OCL is formally defined in [Richters, 2002]. Currently, the 2.0 version of UML is about to be finalized [OMG, 2004] but the language definition will still be informal.

By translating a given UML model into a graph transformation system we provide an integrated formal semantics for a large part of UML. Integrated means that a model in our approach may comprise use case, class, object, statechart and interaction (collaboration and sequence) diagrams. We stick to UML 1.5 but UML 2.0 likewise includes the UML concepts covered by us, albeit some details and the naming have changed in some cases. In particular, collaboration diagrams are called communication diagrams in UML 2.0.

The graph transformation system consists of graph transformation rules and a working graph, which represents a snapshot of the current state (hence called system state) of the modeled system. The system state changes during a run of the system, i.e.

graph transformation rules are applied rewriting parts of the working graph. Using our approach modelers can validate a system model by performing system runs and comparing their expectations with the results of these runs.

In this paper we present the fundamental concept of system states and the translation of a given UML model into a graph transformation system focusing on the rules evolving from statechart and collaboration diagrams. An important aspect for us is that our approach integrates the OCL. OCL expressions that appear in the model are used in rules as well. When applying a rule, these expressions are evaluated.

Several other works can be found, that provide a precise semantics for parts of UML by means of graph transformation. An integrated semantics similar to the one presented here is introduced in [Kuske et al., 2002] for class, object and statechart diagrams. Since this approach does not cover interaction diagrams, it is extended in [Gogolla et al., 2003] adding interaction diagrams on instance level. However, the operations have to be specified by single rules, thus only atomic operations are considered. In addition, the modeller is forced to enrich the UML model with graph transformation by himself, thus that approach is not purely UML conform in contrast to the approach presented in this paper. This work discusses in more detail our approach that has already been presented in [Ziemann et al., 2004] which illustrated the basic concept only by means of an example. In the present paper, we explain the approach in a more general way.

[Varró, 2002] presents a formal semantics for UML statecharts based on a combination of metamodeling and graph transformation. In [Heckel and Sauer, 2001] collaborations are interpreted as visual queries and translated into graph transformation rules using pattern matching. An operation semantics for statecharts based on graph transformation is presented in [Maggiolo-Schettini and Peron, 1994]. In [Kwon, 2000] rewrite rules and its operational semantics are used in order to translate UML statecharts for the purpose of model checking. The Fujaba tool suite [Fischer et al., 1998] automatically generates code from behavioral UML diagrams and additional features using graph transformations that are formulated as story diagrams. Consistency analysis between UML models employing graph transformation can be found in [Engels et al., 2002] and [Tsiolakis and Ehrig, 2000]. In the first work graph transformation is used to refine UML real time models. Their consistency is checked in the semantic domain of CSP. In the second work the consistency between class and sequence diagrams is checked by means of graph transformation.

The structure of the paper is as follows. In the next section an example model consisting of a class, collaboration, and statechart diagram is introduced and explained. Section 3 provides an overview of system states in general. The formal background of our graph transformation approach and the translation of a model into a graph transformation system is explained in Section 4, focusing on the rule generation concerning the sending of messages. The paper closes with a conclusion.

2. Integrated Specification with USE

In order to explain the core mechanism of our approach, we provide a small example model. It only consists of diagrams that are needed for the mechanisms described later on, which concentrate on a central part of our approach handling class, collaboration and statechart diagrams. A proper model would also comprise an object diagram specifying the initial system state and a use case diagram specifying the operations the user can invoke. In the example we model a very basic chat system, resembling the Internet chat system ICQ. Figure 1 depicts the class diagram of this model.

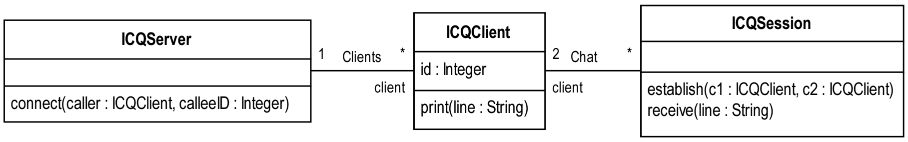

Figure 1: Example class diagram

The model comprises the three classes ICQServer, ICQClient, and ICQSession. An ICQServer can have any number of ICQClients, while the latter can only be a client of exactly one ICQServer. The ICQClient has an attribute id, which uniquely identifies a user operating the client. This id has been created during the registration, which is necessary for users to be able to be part of ICQ chats. This registration is not modeled in our example. The actual chat of exactly two ICQClients is managed by the class ICQSession. Naturally, an ICQClient can take part in more than one chat, therefore it can be connected to any number of sessions.

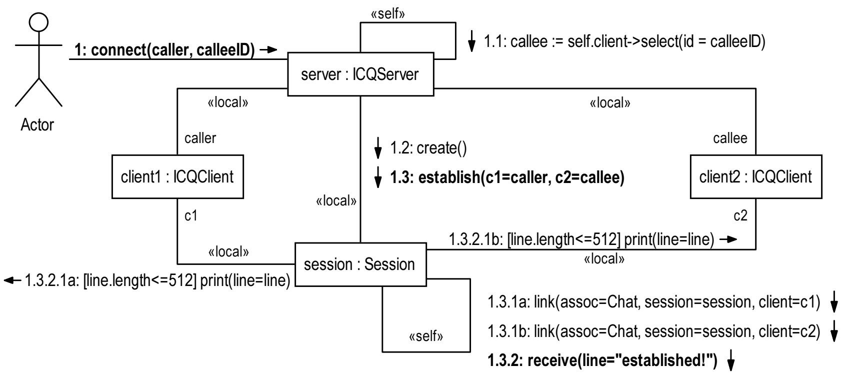

Figure 2: Example collaboration diagram

The operations belonging to the classes ICQServer and ICQSession are specified in the collaboration diagram shown in Fig. 2. A collaboration diagram specifies an interaction of objects by visualizing classifier roles exchanging messages via association roles. Classifier roles are depicted as rectangles containing identifiers and corresponding class names (the base classifier of the classifier role), separated by a colon. Classifier roles represent objects (instances of the corresponding base classes) that play a specific role in the interaction. Association roles are depicted as edges connecting the classifier roles. The association roles may be labeled with a colon followed by the name of an association from class diagram (base association). However, this mechanism is not used in the example. Special stereotypes (text in guillemots) mark association roles that have no base association. It is for instance always possible that a classifier role sends a message to itself, via an association role stereotyped as ≪self≫ (as depicted in Fig. 2 at ICQServer and ICQSession). The stereotype ≪local≫ specifies that the classifier role at one end of the association role represents a local variable with the name of that variable given as role name. Arrows next to association roles depict messages, pointing in the direction of the receiving classifier role. Messages always invoke some kind of operation. Note that the invocation of an operation is not the same as its actual execution. An invocation is regarded as putting an execution request into the queue of the receiving object.

A solid filled arrowhead depicts a synchronous message, i.e., any following message cannot be sent until the operation that has been invoked by the synchronous message has been finished. Asynchronous messages are depicted by stick arrowheads, but our example does not contain asynchronous messages. The nesting of sequence numbers reflects the order of these messages as well as different levels of activation. Increasing numbers on the same nesting level represent successive message calls from the same level of activation, e.g., 1.3 is the successor of 1.2. If messages are on the same level of activation, they are sent from the same classifier role. If a message is the activator of one or more messages, the nesting level of the activated messages is increased, e.g., 1.1 is a message that has been activated by message 1. An operation is specified by a collaboration diagram if there is a message calling the operation and a sequence of messages activated by this message. Thus, the collaboration diagram shows which suboperations the specified operation calls.

The presence of letters in sequence numbers indicate a parallel sending of messages, e.g., the messages 1.3.1a and 1.3.1b are sent in parallel. It is possible to provide an expression in squared brackets depicted behind the sequence number. This expression is called a *guard*, i.e., a boolean expression that has to be evaluated to true for the corresponding message to be actually sent. If the guard evaluates to false, the message will not be sent.

The example collaboration diagram specifies the process of establishing a chat between two chat partners. The situation here is that a number of clients is already connected to the server. At first the user who wants to chat with another user calls the connect operation of the ICQServer (message with sequence number 1). The parameters are the ICQClient caller (representing the user end of the chat communication), and an Integer calleeID containing the unique id (as explained above) of the user with whom the caller wants to establish a chat session. The server then identifies the ICQClient representing the callee by selecting the client of its internal list that has the same id as the one requested by the caller (1.1). It then creates a local session (1.2) and calls the establish operation of that session (1.3), providing the caller and callee clients as parameters. This leads to the call of link operations (1.3.1a and 1.3.1b), linking the two objects representing the clients to that session, thus instantiating the association Chat. These operations are specified to be executed in parallel. Next the session calls its own operation receive (1.3.2) with a constant string "established!" as parameter. This is a message to inform the users represented by the two clients that a connection has successfully been established. This is achieved by a parallel call of the print operation (1.3.2.1a and 1.3.2.1b) of the two clients with the constant string "established!" as parameter. In our example the guard "line.length <= 512" is meant to make sure that only lines containing 512 characters or less may be printed to the clients. The fact that longer lines would not be printed in our model is meant to resemble some kind of protection against molesting and is used to demonstrate the handling of guards in collaboration diagrams.

In the example collaboration diagram the operations connect, establish, and receive are specified. The other called operations (e.g. print) are not specified because it is not shown which suboperations are invoked by them. Note that the operation establish is called with actual parameters (caller and callee) in the context of the connect operation, but the messages activated by establish only refer to the formal parameters of establish (c1 and c2).

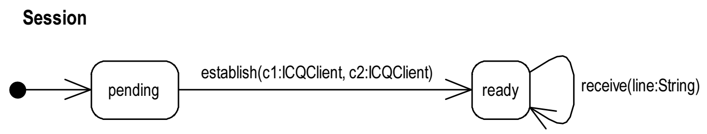

Figure 3: Example statechart diagram

The model is completed by the statechart diagram in Fig. 3. A statechart diagram belongs to a class and comprises states connected by transitions. Transitions are labeled with operation signatures and optional boolean OCL expressions (guards) in square brackets. The shown statechart diagram belongs to the class ICQSession, since this is the only class that actually has relevant states. Initially the state of the session is pending. After the execution of the establish operation, the state changes to ready (the corresponding transition fires). It is now allowed to execute the receive operation on the session, and the statechart does not change its state further on. While collaboration diagrams specify the *calling* of operations by sending messages, the statechart diagram is meant to visualize restrictions of operation *executions*, i.e, in which order the actual operation calls may be handled. If no statechart diagram exists for a class, then any operation belonging to it can be called at any time. But in the case of a present statechart diagram, an operation on an object is only allowed if the object is in a state with an outgoing transition labeled with this operation. In our example, it is not allowed to execute the receive operation if the state of the session is pending. It is also not allowed to execute the establish operation if the state is ready. However, an operation that is not depicted in the statechart diagram of its class may still be executed at any time. If a transition is labeled with a guard, it has to be evaluated to true for the transition to fire. Our graphically simulated system run never violates the restrictions specified in the statechart diagrams.

3. System States

As the name may suggest, the system state represents the internal state of the software system specified by a UML model. We define the abstract syntax of a system state by a class diagram that resembles a part of the UML metamodel. It is shown in Fig. 4.

A concrete system state is then represented by an object diagram that instantiates this class diagram. A formal basis for class diagrams and their instantiation by object diagrams can be found in [Richters, 2002]. A system state can of course be visualized in a more readable way, as long as the contained information is equivalent to the defined abstract syntax.

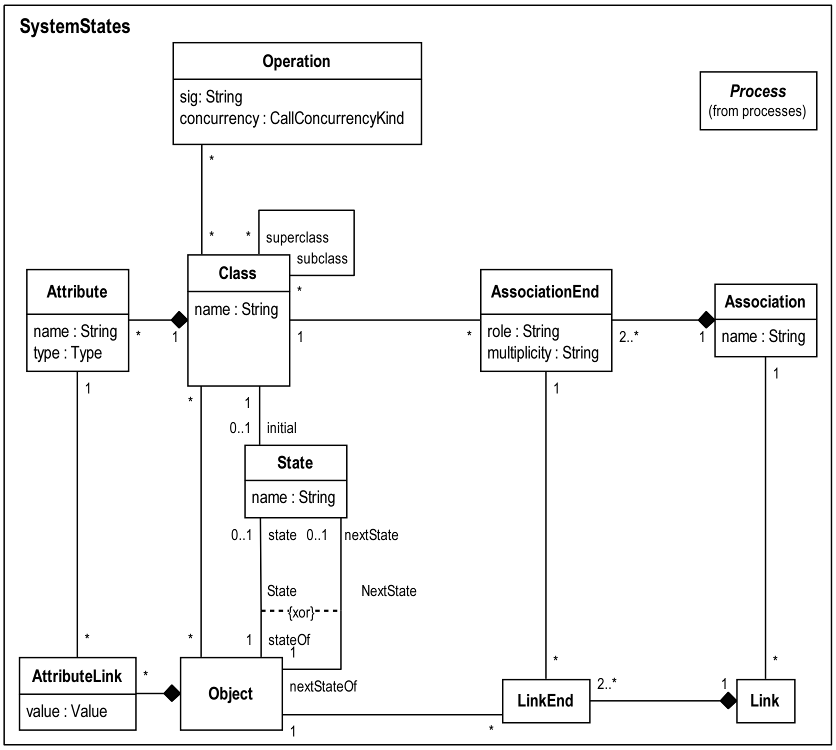

Figure 4: Abstract syntax of system states

Before the evolution of the system starts, the system state resembles the initial object diagram provided by the modeler. It contains nodes of type Object representing the objects that initially exist, connected to nodes of type Class, representing their class. However, it is possible, that the model comprises statechart diagrams specifying initial object states. In this case these states are added to the system state by depicting them as nodes of type State and connecting them to the corresponding object nodes via a State link. The corresponding classes are also connected to their initial states for internal reasons concerning certain graph transformation rules. Further information from the statechart diagrams are not reflected in the system state but rather in the derived graph transformation rules.

Operations that belong to classes are depicted by Operation nodes that are connected to their Class nodes. To be more precise, an Operation node representing an operation of a class is connected to the class and its subclasses. Moreover, an overriding operation is connected to the class the overriden operation is defined for. Operation nodes that are not connected to a Class node represent use cases.

Objects are linked via Link and LinkEnd nodes, that are in turn connected to Association and AssociationEnd nodes. Attributes of objects are represented by AttributeLink nodes that are connected to an Attribute node that declares the attribute in the context of a class.

The system state of the modeled system is changed by processes. There is a number of different process types that are depicted in Fig. 5 and Fig. 6. There are atomic processes and so-called OpCall processes, the latter representing user-defined operations in execution. Processes can have local variables when executed, so they may be connected

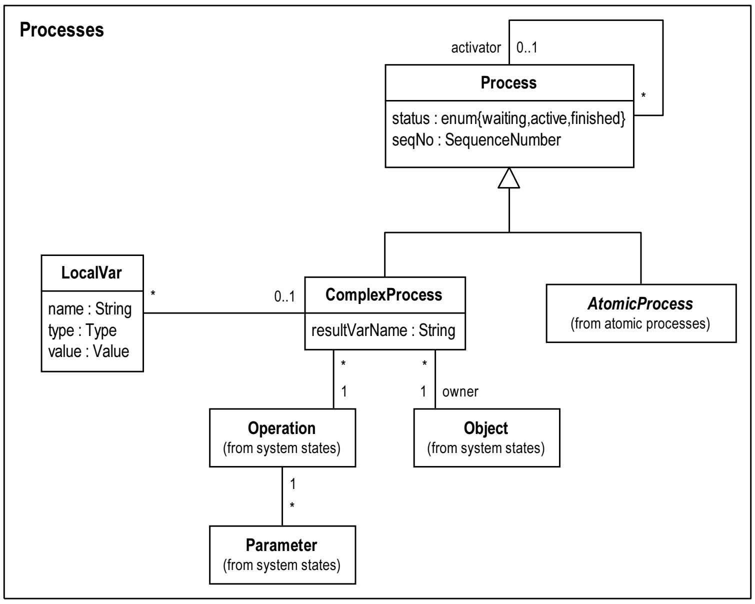

Figure 5: Processes

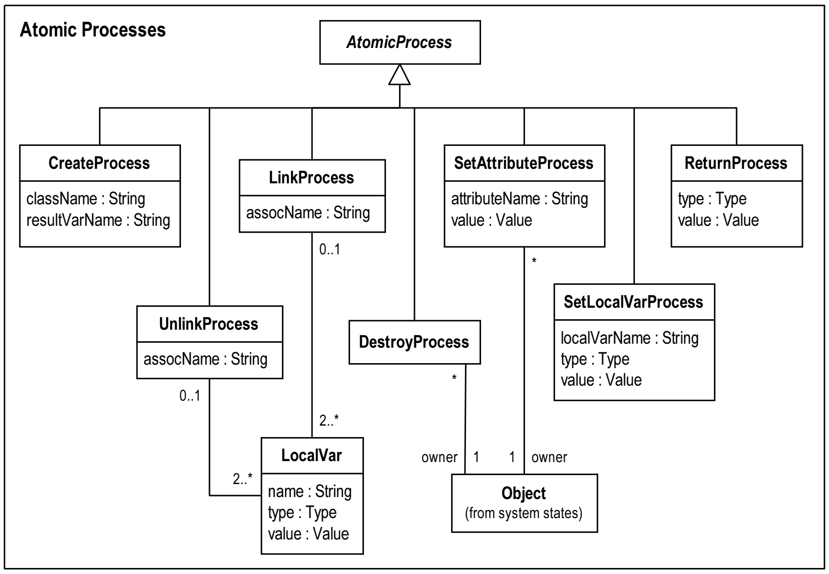

Figure 6: Atomic processes

to LocalVar nodes storing these values. Some process types are connected to an owner object, i.e., the Object node representing the object the operation is called on. Note that the attribute value present in several classes is of type Value, which subsumes all available value types. However integrity constraints that are not covered here ensure that the actual value type corresponds to the type given by the type attribute.

The atomic processes are not connected to Operation nodes, which exclusively represent operations of classes or use cases. Atomic processes represent predefined operations in execution. They comprise Create, Destroy, Link, Unlink, SetAttribute, SetLocalVar and Return processes. A Create process is used to create an object. A Destroy process dispels a given object and is thus connected to the Object node representing the instance to be destroyed. Link and Unlink processes create and destroy links between object nodes. The SetLocalVar and SetAttribute processes change values of local process variables and object attributes, respectively. A Return process finishes its activator process and handles a potential return value.

A subprocess, i.e., a process that has been called by another process (the activator), is connected to the calling process. This activator structure represents the activator tree in the UML metamodel for interaction diagrams.

All process types have a sequence number and a status as common attributes. The sequence number is taken from the specifying collaboration diagram of the model. The status can be #waiting, #active or #finished. This will be explained in more detail in Sect. 4.

4. Rules for Sending Messages

In this section we describe how to construct rules originating in collaboration and statechart diagrams. The description is general but we will also often refer to the chat example introduced earlier. A collaboration diagram specifies operations of classes by showing messages that are sent by the operations in a specific order. The collaboration diagram in Fig. 2 specifies three operations: connect(caller:ICQClient, calleeID:Integer) of class ICQServer, establish(c1:ICQClient, c2:ICQClient) and receive(line:String) of class ICQSession.

A message invokes a user-defined operation or a predefined operation like setting an attribute value, setting a local variable or creating an object. A user-defined operation is an operation that is declared in the class diagram for a specific class. For example, the message 1.2 calls a predefined operation while the message 1.3 calls a user-defined operation.

4.1. Graph Transformation

A graph transformation system consists of a working graph and a set of rules which rewrite parts of this graph when applied. We use the algebraic graph model for attributed, directed and labeled graphs and their transformations (cf. [Löwe et al., 1993], [Rozenberg, 1997]). A graph transformation rule consists of a left-hand and a right-hand side, both being system states as explained earlier. Usually the rules are notated in the following way. Nodes that should be preserved during the rewriting occur in both sides of the rule. They are identified by identifiers in the upper compartment, in front of the colon. Nodes that only occur in the left-hand side are deleted while nodes that only occur in the right-hand side are added to the working graph. Negative Application Conditions (NAC) are specified as graphs that extend the left-hand side in order to specify a situation that is not wanted in the working graph, i.e., if such a situation can be found, the rule cannot be applied. Application conditions in the form of boolean OCL expressions (including invocations of side effect-free operations) may be used as well to restrict the application of a rule in certain situations. These expressions are evaluated in an analogous way to OCL expressions in [Richters, 2002], since the system state graph represents a special object diagram, which in turn corresponds to a formal system state as explained in [Richters, 2002]. Naturally, the evaluation takes place in the context of this system state rather than the system state graph. Variables representing attribute values in the usual way can also be used in both sides of a rule. These variables can also be employed in OCL expressions in the right-hand side of a rule in order to calculate new attribute values. Thus the right-hand side of a rule actually differs from the system state defined earlier in that attributes may hold general OCL expressions and not only constants.

4.2. The Basics

In our system state, the sending of a message corresponds to the creation of a process node with status #waiting. When a waiting process is activated by a rule, its status changes to #active. An active process represents an operation in execution. In the following, we simply say that a rule sends a message (or invokes an operation) if it creates a process node.

The set of rules handling the invocation of the suboperations of a user-defined operation is said to execute the operation. We concentrate on these rules here. So, let us assume that there is (among many other nodes) a waiting process of a user-defined operation in the system state (technically, there is a ComplexProcess node with status = #waiting that is connected to a node).

We then need rules for sending the messages as specified in the collaboration diagram for the operation. In each rule, we have to decide under which circumstances the rule shall be applicable and to which target the message shall be sent.

In the simplest case (if there is no statechart diagram for the class of the sending classifier role) we only need one single rule for sending a message. The framework of the rule differs depending on the kind of message. In the following let us consider a message that calls a user-defined operation.

We will describe the construction of a rule in a general way. For this purpose, consider Fig. 7 that shows a class diagram scheme. There could be more classes, associations, attributes and operations but we need only the shown features here. Figure 8 shows a collaboration diagram scheme that specifies the operation opname_a of class Classname_a. This scheme again shows only the features we need to explain the rule construction; the message 1.x (where x is a natural number) is only one message among many others that are not shown here. The scheme for the general framework of the rule sending this message is depicted in Fig. 9. For space reasons, left-hand and right-hand side of the rule are displayed in a single graph: The bold edges and nodes are only present in the right-hand side. Crossed out elements are only present in the left-hand side.

In the **left-hand side** we have a ComplexProcess node representing the activa-

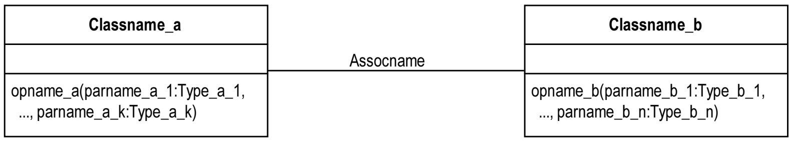

Figure 7: Class diagram scheme

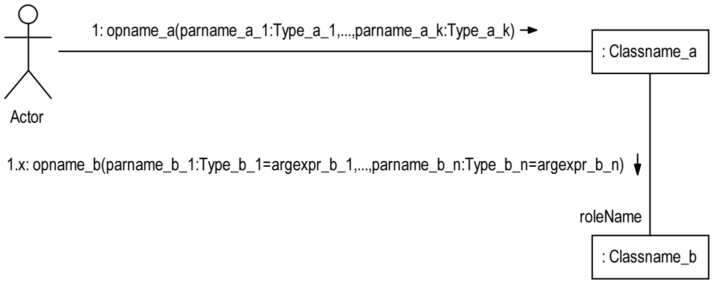

Figure 8: Collaboration diagram scheme that specifies opname_a **of class** Classname_a

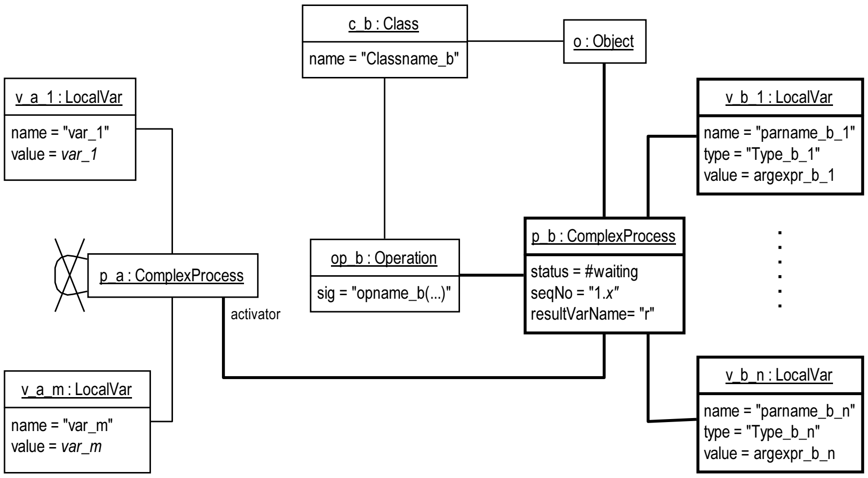

Figure 9: Scheme of the rule that sends the message 1.x**.**

tor process. We assume that in the preceding rule application a proper process node has been marked with a loop. Thus, it is not necessary to connect this node to an Operation node now. This allows for recursive operation calls in spite of injective matching. A proper activator process is connected to the operation op_a of class c_a and has the status #waiting if the message we want to send by this rule has no predecessors in the collaboration diagram, and #active otherwise.

In addition, we have an Operation node op_b connected to a Class node c_b. This represents the operation to be executed and its class, so the name attribute of op_b and c_b are set accordingly (e.g., to establish resp. ICQSession). An NAC not shown here ensures that dynamic dispatching works correctly. Finally, there is an Object node o connected to c_b. That is the object the operation will be invoked on.

In the **right-hand side**, all the mentioned nodes are present as well, because corresponding nodes in the system state graph are not supposed to be deleted when applying the rule. The rule is supposed to create a new ComplexProcess node, so we add a ComplexProcess node p_b that is connected to o and to op_b. The activator edge connects p_b with p_a, identifying p_a as the activator of p_b. Its status is set to #waiting, indicating that the message is sent and the operation waits to be executed. The seqNo and (if present) the resultVarName is set according to the collaboration diagram (sequence number in front of the first ":", result variable name in front of the assignment symbol ":="). In our example, the message 1.1, which calls a predefined operation for setting a local variable, has a result variable named callee.

For each parameter par_b_i of op_b we add a LocalVar node v_b_i with the same name and type. Each v_b_i is connected to p_b. So the newly created process has a local variable for each parameter. The value attributes of the v_b_i,...,v_b_n are set according to the arguments of the message as given in the collaboration diagram.

4.3. Special Considerations

Variables in OCL Argument Expressions. If a called operation has parameters, they are given as OCL argument expressions in the collaboration diagram. These expressions can be used unchanged in the rules as described above. However, such an OCL expression can contain variables that have to be bound when applying the rule and evaluating the expression. The message can contain only variables that are known in the context of the activator process.

Let *var_1,...,var_m* be the free variables contained in the argument expressions. Then for each *var_i* we add a LocalVar node v_a_i in both sides of the rule, where the name attribute of v_a_i is set to the constant var_i and the value attribute is set to a variable that is also named *var_i*. All these LocalVar nodes are connected to p_a. When applying the rule with any match, the match binds each *var_i* to a concrete value, so the expressions can be evaluated and the result is inserted into the system state graph.

Synchronous and Asynchronous Predecessors. If the rule sends a message that has one or more predecessors in the collaboration diagram, the rule has to be applicable only after (1) the predecessor messages are sent and (2) the operations invoked by the synchronous predecessors have been finished. We achieve this by adding a process node for each predecessor and connecting it to the activator process p_a with the activator edge in the left-hand side of the rule. The seqNo attribute is set according to the collaboration diagram to identify the predecessor. If the predecessor is synchronous, status is set to #finished. On the other hand, if it is asynchronous the attribute status is not set because the existence of the process is enough then (i.e., the fact that the message has been sent). Finished processes that are not longer needed as predecessors are removed by a special garbage collection rule that is not presented here.

Sending via Local or Self Association Roles. If the message under consideration is sent via an association role with the stereotype ≪local≫, it means that the receiver of the message is referred to in a local variable of the activator process. The name of this variable is given as the role name of the receiving classifier role. So we add a LocalVar node in both sides of the rule and connect it to p_a. Its name is set to the role name from the collaboration diagram and its value is set to the receiving object o. In this way we ensure that by applying the rule the process is attached to the object that is stored in the specified local variable.

If the message is sent via an association role with the stereotype ≪self≫, it is sent to the object the activator process is currently running on. That means that we additionally insert an owner edge from the receiving object o to p_a.

Sending via an Ordinary Association Role. When a message is sent via an ordinary association role, i.e., an association role without any stereotype, the modeller has not yet specified how to determine a concrete receiver object. The only demand is that it is linked with the sender object with a link compliant to the association role. This is ensured in the rule by additional nodes and connections.

It often happens that in a collaboration diagram a classifier role receives more than one message during the execution of the operation specified by the collaboration. However, if two or more messages are sent via an ordinary association, we so far do not ensure that the messages are sent to the same object because the receiving Object nodes in the different rules can be matched to different objects in the system state graph. We ensure this in the following way: The rule that sends the first of several messages via an ordinary association rule stores the chosen receiver in a new local variable of the activator process. Rules that send messages via the same association role later in the interaction do this in the same way in which messages are sent via a local association role.

Sending Parallel Messages. So far, a rule created only one process, i.e., sent one message. When several messages are specified to be sent in parallel in the collaboration diagram, all those messages are sent by one single rule. For each classifier that receives one of these messages, there is an Object node in both sides of the rule. Two messages sent to the same classifier role result in two processes attached to the same Object node. The procedure for the single messages is as described above.

Guards. A guard is a boolean OCL expression that can be written in front of a message in the collaboration diagram (the messages 1.3.2.1a and 1.3.2.1b have a guard in our example). If it evaluates to false, the message shall not be sent. However, in our system state graph we need the information that this message is handled in case there are other messages waiting for this message to be handled. For this reason, we always allow the corresponding rule for sending the message to be applied. But if the guard evaluates to false the status of the created process is set to #finished. Thus, the corresponding operation is not executed but the following messages can proceed. This is done by setting the status of the process to be created in the right-hand side to the OCL expression "if G then #waiting else #finished" where G is the guard expression given in the collaboration diagram in front of the message. And again, to be able to evaluate the OCL expression when applying the rule, we have to add LocalVar nodes to the rule as described in subsection 4.3.

States. A statechart diagram can be given for a UML class to specify in which order operations on an object of the class may be executed. In our chat example, the establish operation is allowed to execute only when the session is in the state pending. We assume that operations of a class that do not occur in the statechart for the class are allowed to be executed at any time. The statechart also specifies in which state the object is after an operation has been finished. In our example, the state of a session changes to ready as soon as the establish operation has been finished.

Because the statechart specifies whether an operation is allowed to execute, we have to modify the rule that starts the operation by sending the first message (or several parallel first messages). Let us assume we construct the rules that send messages according to a collaboration diagram for an operation of a class. Furthermore, let us assume that there is a statechart for this class and there are n transitions labeled with the operation. Then we split the rule that sends the first message(s) of the collaboration diagram into n versions that differ from each other by the State nodes attached the Object node and by an optional application condition. This application condition holds the OCL expression given by the optional guard of the considered transition. For this reason the rule is only applicable if the guard evaluates to true.

A State node is connected to an Object node either by a state edge or by a nextState edge. Only one edge is allowed at a time. When an object is connected to a state by a nextState edge, it is in no regular state but it is "between" two states. The state node it is connected to denotes the state the object will have soon.

Let one of the n mentioned transitions lead from a state A to a state B. In the corresponding version of the rule, we add an Object node o_a and connect it to the activator process p_a in both sides of the rule with an owner edge. Then we add a State node to the left-hand side with name = A and connect it to the Object node o_a with a state edge. In the right-hand side, we add another State node with name = B and connect it to o_a with a nextState edge.

So, this rule cannot be applied if the object to which the o_a node is matched to in the system state graph is not in state A. On the other hand, if the rule can be applied, the State node is deleted and the object is instead connected to its next state. The "next state" becomes the "current state" by replacing the nextState edge by a state edge. This is done by the Return rule that is not described here.

5. Conclusion and Future Work

We have shown a translation from a UML model into a graph transformation system in order to give a precise formal semantics for a large part of UML. In this paper, we concentrated on the construction of graph transformation rules that represent the sending of messages as specified in a collaboration diagram. At the beginning, an example model has been introduced focusing on a collaboration and a statechart diagram. Then, we have described the concept of a system state being a graph representing the current state of the modeled system and being transformed by graph transformation rules as the system evolves. In the central section, we have described in a general way how to construct a rule that corresponds to the sending of a message specified in a collaboration diagram. This description has focused on messages that invoke an operation of a class in contrast to messages that invoke a predefined operation like setting an attribute value.

Currently a tool is developed that implements the approach discussed in this paper. The purpose of this tool is to visualize the evolution of a system specified in UML before the system is actually implemented. Loaded with a UML model consisting of class, use case, object and collaboration diagrams, it generates the initial system state and the graph transformation rules. The user can then view the evolution of the system by applying rules step by step. The rules are chosen by the application based on the user's selection of use cases (together with parameter values) or waiting processes. It is also possible to evaluate OCL expressions in the current system state. Using our tool, the user does not need to know the details of our approach. Only the usage of the considered UML diagrams has to be known.

Future work comprises the completion of the tool, including a user-friendly GUI. Then our approach can be evaluated in case studies that provide feedback on its usefulness. Furthermore, we will investigate how the approach can be extended with UML features not covered yet. This includes additional features of already covered diagram types but also entire diagram types like activity diagrams. Another important issue that has to be studied more extensively is how our approach allows to assert properties of UML models by examining the resulting graph transformation system.

References

Engels, G., Heckel, R., Küster, J. M., and Groenewegen, L. (2002). Consistency-Preserving Model Evolution through Transformations. In Jézéquel, J.-M., Hussmann, H., and Cook, S., editors, *UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings*, volume 2460 of *LNCS*, pages 212–226. Springer.

Fischer, T., Niere, J., Torunski, L., and Zündorf, A. (1998). Story Diagrams: A new Graph Transformation Language based on UML and Java. In Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G., editors, *Proc. Theory and Application to Graph Transformations (TAGT'98), Paderborn, November, 1998*, volume 1764 of *LNCS*. Springer.

Gogolla, M., Ziemann, P., and Kuske, S. (2003). Towards an integrated graph based semantics for UML. In *Graph Transformation and Visual Modeling Techniques (GT-VMT 2002)*, volume 72 of *ENTCS*.

Heckel, R. and Sauer, S. (2001). Strengthening UML Collaboration Diagrams by State Transformations. In Hussmann, H., editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, Proceedings*, volume 2029 of *LNCS*, pages 109–123. Springer.

Kuske, S., Gogolla, M., Kollmann, R., and Kreowski, H.-J. (2002). An Integrated Semantics for UML Class, Object, and State Diagrams based on Graph Transformation. In Butler, M. and Sere, K., editors, *3rd Int. Conf. Integrated Formal Methods (IFM'02)*, volume 2335 of *LNCS*, pages 11–28. Springer.

Kwon, G. (2000). Rewrite rules and Operational Semantics for Model Checking UML Statecharts. In Evans, A., Kent, S., and Selic, B., editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 528–540. Springer.

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

Maggiolo-Schettini, A. and Peron, A. (1994). Semantics of full statecharts based on graph rewriting. In Schneider, H. and Ehrig, H., editors, *Proc. Graph Transformation in Computer Science*, volume 776 of *LNCS*, pages 265–279. Springer.

OMG (2003). *OMG Unified Modeling Language Specification, Version 1.5, March 2003*. Object Management Group, Inc., Framingham, Mass., Internet: `http://www.omg.org`.

OMG (2004). UML 2.0 superstructure final adopted specification. Technical report, Object Management Group. `http://www.omg.org/cgi-bin/doc?ptc/2003-09-15`.

Richters, M. (2002). *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14.

Rozenberg, G., editor (1997). *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. World Scientific, Singapore.

Tsiolakis, A. and Ehrig, H. (2000). Consistency Analysis of UML Class and Sequence Diagrams using Attributed Graph Grammars. In Ehrig, H. and Taentzer, G., editors, *Proc. of Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems, Berlin, March 2000*. Technical Report no. 2000/2, Technical University of Berlin.

Varró, D. (2002). A formal semantics of UML statecharts by model transition systems. In Corradini, A., Ehrig, H., Kreowski, H.-J., and Rozenberg, G., editors, *Graph Transformation. First International Conference, ICGT 2002, Barcelona, Spain, October 2002, Proceedings*, volume 2505 of *LNCS*, pages 378–392. Springer.

Ziemann, P., Hölscher, K., and Gogolla, M. (2004). From UML models to graph transformation systems. In Minas, M., editor, *Preliminary Proceedings: Workshop on Visual Languages and Formal Methods*.