Available online at www.sciencedirect.com
SCIENCE DIRECT®

ELSEVIER Electronic Notes in Theoretical Computer Science 127 (2005) 17–33

Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

From UML Models to Graph Transformation Systems

Paul Ziemann[1] Karsten Hölscher[2] Martin Gogolla[3]

Department of Computer Science
University of Bremen
Bremen, Germany

Abstract

In this paper we present an approach that allows to validate properties of UML models. The approach is based on an integrated semantics for central parts of the UML. We formally cover UML use case, class, object, statechart, collaboration, and sequence diagrams. Additionally full OCL is supported in the common UML fashion. Our semantics is based on the translation of a UML model into a graph transformation system consisting of graph transformation rules and a working graph that represents the system state. By applying the rules on the working graph, the evolution of the modeled system is simulated.

Keywords: Graph transformation, UML semantics, validation, CASE tool

1 Introduction

Today the Unified Modeling Language (UML) is widely accepted as a standard for modeling object-oriented software systems. UML is a graphical language providing different diagram types for describing particular aspects of software artifacts. The syntax of these diagrams is defined by means of a metamodel in [12], notated as class diagrams. However this approach is semi-formal, since the class diagram itself is defined in a cyclic way by the metamodel.

[1] Email: ziemann@informatik.uni-bremen.de
[2] Email: hoelscher@informatik.uni-bremen.de
[3] Email: gogolla@informatik.uni-bremen.de

doi:10.1016/j.entcs.2004.10.025

Furthermore the semantics of UML diagrams is only expressed in natural language. The graphical notation is enhanced by the Object Constraint Language (OCL), which permits to formulate constraints in a textual way that cannot be expressed by the diagrams. OCL is again semi-formally defined in [12]. A formal syntax and semantics for UML class diagrams as well as OCL has been introduced in [13], which is also included in the accepted OCL 2.0 OMG submission [1].

In this paper we present an integrated formal semantics not only for class diagrams but for further basic diagram types: use case, object, statechart and interaction 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 new integrated semantics is formalized employing the concepts of graph transformation, which is a well-developed field (cf. [15], [4], [5]). We are not aware of a formal approach handling this collection of UML diagrams, in particular the formal incorporation of use cases is new (in [17] use cases are described precisely by so-called operation schemas including OCL pre- and postconditions but the connection to other UML diagrams is left open).

Our approach provides a framework for an automatic translation of a UML model into a graph transformation system. The UML model may consist of the mentioned diagram types and can include OCL expressions. The graph transformation system comprises a set of graph transformation rules and a so-called working graph, hence called system state graph. As the name may suggest, the system state graph represents the current state of the modeled system. The graph transformation rules modify this state step by step, thus simulating a run through the modeled system.

In contrast to most work on graph transformation, we employ an enhanced approach, which allows OCL expressions in rules. We combine the advantages of two worlds: the operational graph transformation world and the logic-based OCL world. On the one hand graph transformations allow to handle complex issues by depicting and modifying them using more intuitive graphical representations. On the other hand, although it is theoretically possible to represent every aspect in the graphical structure, the additional power to use OCL as a textual notation leads to the benefit of even more compact graphs in most cases. In our approach OCL expressions navigating in the current system state are used as application conditions, which decide whether a certain rule may or may not be applied. Furthermore OCL is used in attribute expressions in the right-hand side of graph transformation rules. The modeler can also utilize OCL for querying the current state of the modeled system.

Our approach provides an integrated formal semantics for a large part of UML. As no formal semantics is given for the UML, our approach relies on a number of assumptions on how the diagrams could be used in practise and integrated in a useful way. The precise semantics is a solid basis for further work. For example, the representation of a UML model as a graph transformation system is used here to validate the system before actually implementing it. This is done by comparing system behavior with the expectations of the modeler. The benefit of using graph transformations in this context obviously is the close proximity of the simulated system run to the actual model. This proximity allows for fewer assumptions regarding the semantics of the model as for instance code generators have to make. It also allows for an easier handling of future extensions and changes regarding these assumptions, since only the structure of the generated rules has to be changed in these cases. Currently, a prototypic validation system is being implemented for our approach which generates the graph transformation rules for a given model and allows to interactively execute and visualize the modeled system. Our approach can be used by a modeler in an early stage of a software development process in order to get a flair for the newly designed system.

There are several other works aiming at defining a semantics for parts of UML using graph transformation. In [10], an integrated semantics is given for a large part of UML. However, interaction diagrams and OCL are not considered. Their approach is extended with interaction diagrams on instance level in [8]. Operations are still specified by single rules, that is, all operations have to be atomic. More efforts exist considering isolated parts of UML. In [9], collaborations are translated into transformation rules, where collaborations are interpreted as visual queries using pattern matching. A formal semantics for UML statecharts is presented for example in [19]. The Fujaba tool suite [7] supports graphical object-oriented software design and automatic code generation from story diagrams. These diagrams combine behavioral UML diagrams and additional features. Additional approaches for consistency analysis of UML models can be found. In [6], given UML real time models are refined using graph transformation rules and their consistency is checked in the semantic domain of CSP. [18] addresses the consistency analysis between UML class and sequence diagrams based on graph transformation.

The structure of the rest of this paper is as follows. In the next section the covered UML features of the model are presented and explained using a simple example. Section 3 deals with the detailed description of the system state concept. The translation of the model into a graph transformation system is presented in Sect. 4 by example. The fundamental architecture of the prototypic implementation is presented in Sect. 5. The paper closes with a conclusion in Sect. 6.

2 Covered UML Features

We cover the following UML features: use case, class, object, statechart, and interaction diagrams (collaboration and sequence diagrams) and last but not least full OCL.

We support class diagrams for defining the structure, and interaction diagrams for realizing operations declared in the class diagram. An interaction diagram contains a sequence of messages calling either an operation of a class that in turn is realized by an interaction diagram or calling a predefined functionality like creating an object or setting an attribute value.

Use cases are likewise realized by interaction diagrams. A use case resp. its realization states which operations are called by an actor and in which order this is done. Statechart diagrams specify the order in which operations on an object may be executed. The kind of statechart diagram we support are so-called protocol machines, i.e., statechart diagrams which do not need actions on transitions. Object diagrams are used to specify the system state to start the evolution with and to represent part of the current state of the system.

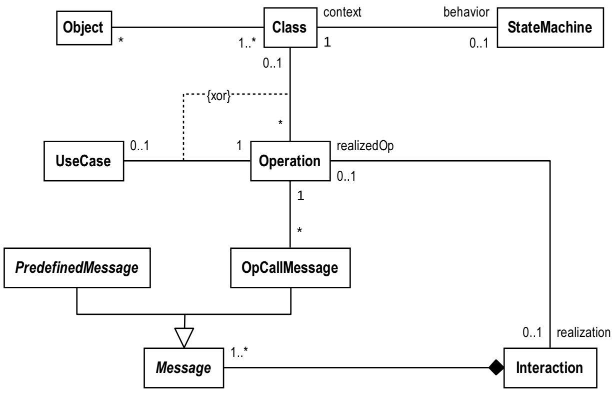

Figure 1. Connection between central modeling concepts

Fig. 1 gives an overview of the connections between the central concepts. We consider one class diagram and one use case diagram. Each class has zero or more operations. A use case is associated with exactly one operation that is not associated with a class. Each operation is realized by an interaction specified in an interaction diagram. An interaction contains messages, which are either predefined (for creating an object or setting an attribute value) or which call an operation of a class. For each class there can be one state machine specified in a statechart diagram. The object diagram instantiates the class diagram. We illustrate the usage and interplay of the diagrams by an example UML model for a digital clock. This a rather simple example; a more complex model fitting our approach can be found at [20].

Fig. 2 shows a use case diagram containing three use cases. The actor can get the time, or set the hours or minutes of the clock. The two use cases for setting the time have a parameter for the hour resp. minute the actor wants to set.

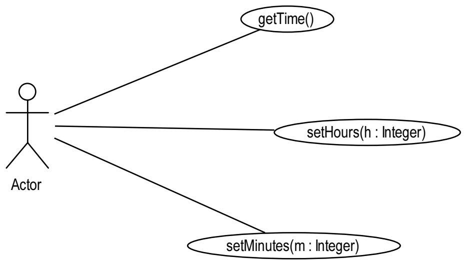

Figure 2. A use case diagram

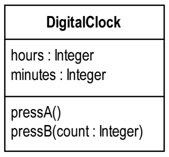

Figure 3. A class diagram

The class diagram in Fig. 3 declares the properties of our clock. It has one attribute holding the hours and one holding the minutes. There are two operations on the clock: pressA() for pressing the A button and pressB() for pressing the B button. In this simple example we have only this single class, however multiple classes with associations and inheritance are supported in our approach.

The statechart diagram in Fig. 4 specifies the states a clock can be in: display, setMinutes and setHours. The initial state points to the state display, which means that once a clock object is created it is in state display. It is also specified here that executing the operation pressA() is allowed in every state and how the state is changed by doing so. The operation pressB() is allowed only in the states setMinutes and setHours and does not change the state.

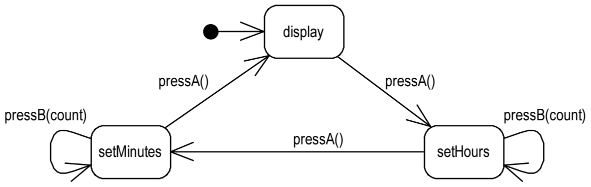

Figure 4. A statechart diagram for the class DigitalClock

The collaboration diagram shown in Fig. 5 realizes the use case setHours by specifying the messages the actor can send. In this case we have three

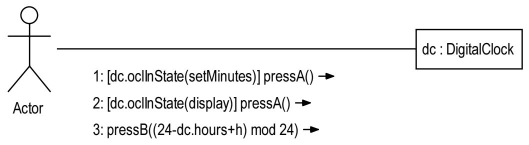
Actor
dc : DigitalClock
1: [dc.oclInState(setMinutes)] pressA()
2: [dc.oclInState(display)] pressA()
3: pressB((24-dc.hours+h) mod 24)

Figure 5. A collaboration diagram realizing the use case setHours(h : Integer)

OpCallMessages as refered to in the metamodel in Fig. 1. The sequence numbers at the beginning of the messages specify the order in which they are sent. The OCL guard in square brackets has to be fulfilled to send the message. Finally, the arrow at the end of a row is used in UML to specify the direction of the message. The type of the arrowhead indicates whether the message is synchronous (filled solid arrowhead) or asynchronous (stick arrowhead). We only consider synchronous messages, that is, before a message is sent it has to wait until the functionality invoked by the preceding message(s) has finished. In Fig. 5, the messages are ordered in a sequence. In the example, at first, the actor has to press the A button once or twice so that the clock is in the state setHours. Then she presses the B button the required number of times, which is represented by giving an appropriate parameter to the pressB operation. The messages are sent to a classifier role representing a digital clock.

The operation pressB(count : Integer) is realized by the collaboration diagram in Fig. 6. When the operation is called on a digital clock, a message depending on the state is sent for setting the attribute. Then, the operation is called recursively if count is greater than zero. These messages are sent from the clock that received the message to itself via a «self» association role. This kind of association role does not need to have a corresponding association in the class diagram.

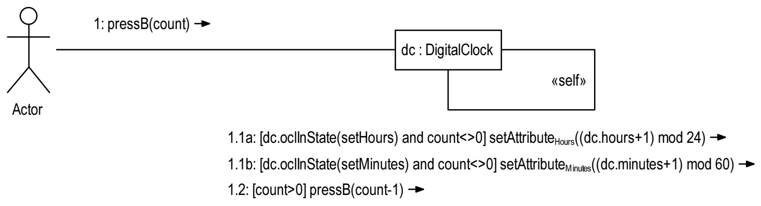
Actor
1: pressB(count)
dc : DigitalClock
«self»
1.1a: [dc.oclInState(setHours) and count<>0] setAttribute_Hours((dc.hours+1) mod 24)
1.1b: [dc.oclInState(setMinutes) and count<>0] setAttribute_Minutes((dc.minutes+1) mod 60)
1.2: [count>0] pressB(count-1)

Figure 6. A collaboration diagram realizing the operation pressB()

The object diagram in Fig. 7 depicts the initial system state of the system: there is one digital clock with its attributes set to zero.

Note that sequence and collaboration diagrams are based on the same information in the metamodel of UML 1.5 and thus are semantically equivalent (cf. [2], pages 249–250). It is even possible to convert one diagram type into the other without loss of information [3]. However, in the concrete syntax of sequence diagrams the association roles are not visualized. If the association

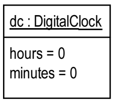

Figure 7. An object diagram

roles were nevertheless included in a sequence diagram, it could also be used instead of a collaboration diagram in the model.

In the next section, we describe the components of the graph representing the state of the modeled system.

3 System States Including Processes

Technically a system state is represented as a directed and labeled graph with attributed nodes and edges (cf. e.g. [11]). To begin with, a system state graph contains attributed objects and links connecting them. So far this graph can be regarded as an object diagram. However, a system state contains two more important concepts: (1) object states, which are attached to objects according to the statechart diagrams, and (2) processes, which represent the actual execution of operations. The abstract syntax of system states is shown in Fig. 8 by means of a metamodel.

The upper part of the diagram represents information from the class and use case diagram (for each use case there is an operation that is not linked to a class). Objects are connected to their classes (their primary class and all its superclasses). Correspondingly, all the other nodes on instance level (Link etc.) are connected to the element on specification level (Association etc.). In order to determine the initial state of an object during the execution of operations, a node representing this state is connected to the corresponding class as specified by an optional statechart diagram. Objects can be connected to a state, representing either its current state or the state the object will be in after completing a currently firing transition. Note that the state of an object is not necessarily related to the configuration of its attribute values but that it can be considered as an additional feature.

The existence of a process in the system state implies that some kind of functionality has been asked for earlier. Suitable graph transformation rules can then be applied to simulate this functionality. These rules also have to respect that specific object states are required according to the statechart.

All kinds of processes have a status and a sequence number and can have an activator process. The status can be waiting, active or finished. An OpCall-Process is connected to an operation, several local variables and an object it is running on (the owner of the process). Fig. 9 and 10 show the different kinds of

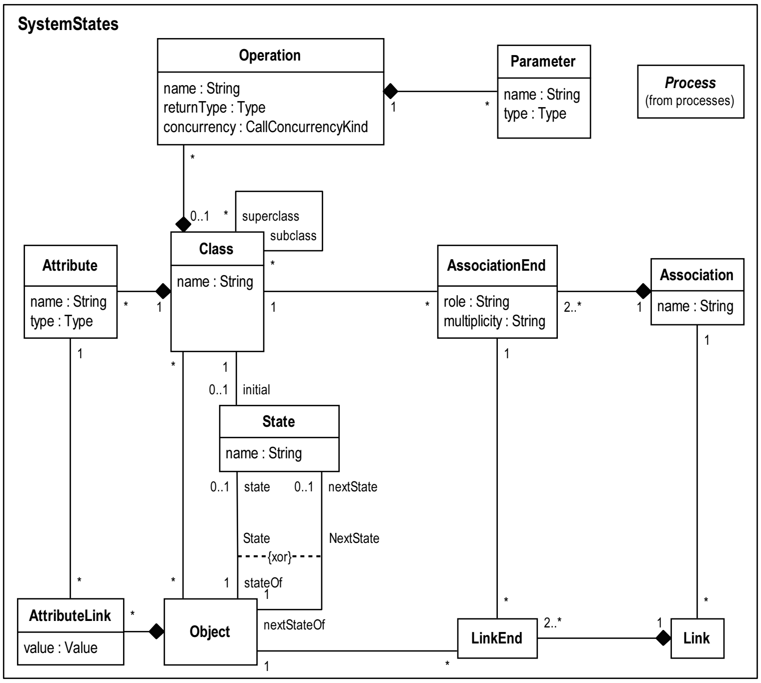

Figure 8. Abstract syntax of system states

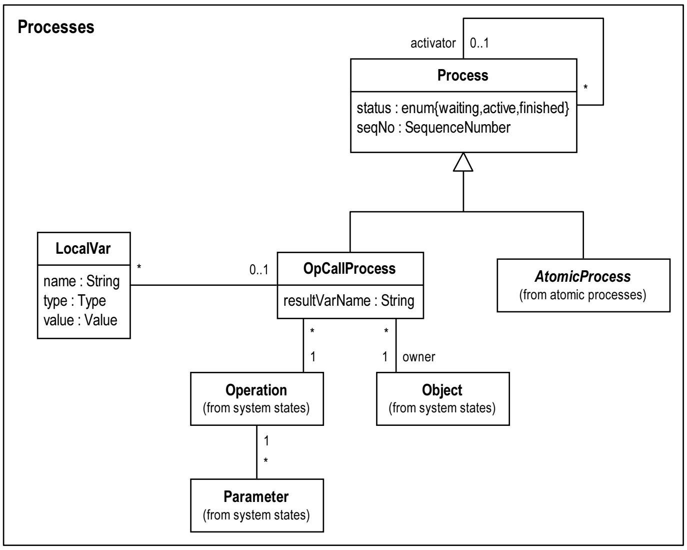

Figure 9. Kinds of processes

processes. Each kind has its special attributes and associations. In particular, Fig. 10 shows the atomic processes corresponding to the predefined messages mentioned earlier. They are called atomic because they do not activate other processes.

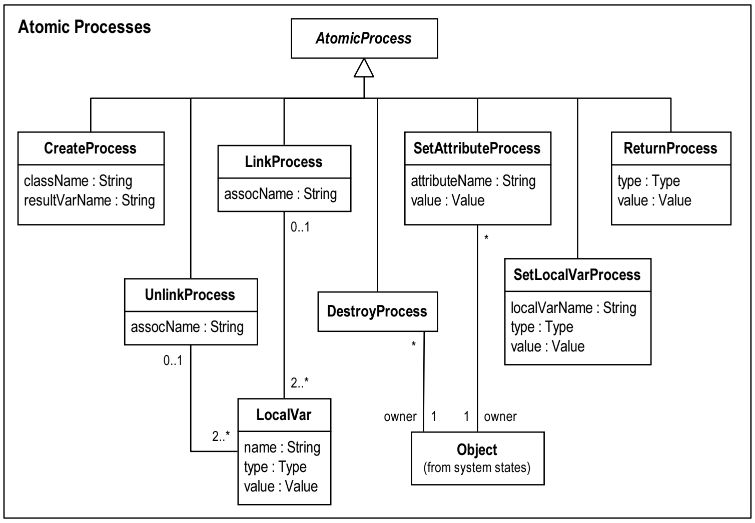

Figure 10. Atomic processes

To briefly illustrate the concept of a process in this context let us consider an OpCallProcess by example. An OpCallProcess is associated with an operation of a class. Let us now assume that an interaction diagram defines that the effect of this operation is exactly to call another operation. Then there would be a rule creating another process node, that is associated with the operation to be called. The system state also includes the information that the first process is the activator of the second one.

4 Translation into a Graph Transformation System

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. e.g. [11]). Roughly speaking a graph transformation rule consists of a left-hand side and a right-hand side. The left-hand side specifies the part of the working graph that has to be changed and the right-hand side specifies these changes. Nodes that should be preserved during the rewriting have to occur in both sides of the rule. 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

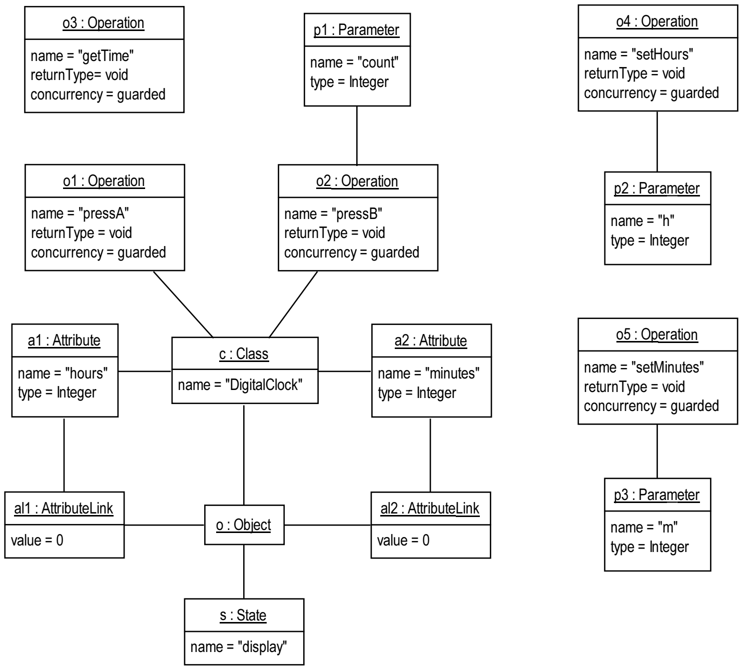

Figure 11. Start system state for the clock example

graph. Every node is marked with an identifier that is notated in the upper compartment right before the colon. Nodes with same identifiers in both sides are thus preserved. Negative application conditions (NAC) may be used as well. They are denoted 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 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 [13], since the system state graph represents a special object diagram, which in turn corresponds to a formal system state as explained in [13]. 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.

As start graph (or start system state) we choose the object diagram the user has delivered for this purpose and attach the initial states to the objects and classes in case there is a statechart diagram for the classes. We then have a system state without processes. Fig. 11 shows the initial system state for our clock example. It is depicted as an instance of the metamodel presented in Fig. 8. We could also have used some concrete syntax to hide the class, attribute, operation and parameter nodes as it is usually done with object diagrams, but here we chose this more abstract representation so that we can better describe how the rules on the system state work.

The system state contains all classes, operations and attributes from the class diagram and also an operation (possibly with parameters) for each use case. In the example, there is one object with two attribute links that is connected to the class and to a state.

Basically we need two kinds of rules: Rules that depend on the given model and rules that do not, i.e., predefined rules. The following two subsections describe these rules and how to construct them.

4.1 Rules Depending on the Model

The initial system state does not contain any processes, i.e., there is no operation that is called and waiting to be executed. This is what the use cases are needed for: For every use case we construct a rule that adds an OpCallProcess node with local variables for holding the arguments where necessary.

Fig. 12 shows the rule for the use case setHours. The rule creates a new OpCallProcess connected to the Operation with the name setHours. The status is set to waiting and the sequence number is set to 0. Because the operation setHours has a parameter h of type Integer, a corresponding local variable node is created and linked with the new process. Its value is set to x, which is a free variable. Thus, this rule is a parameterized rule that needs an assignment that binds x to an actual value before it can be applied.

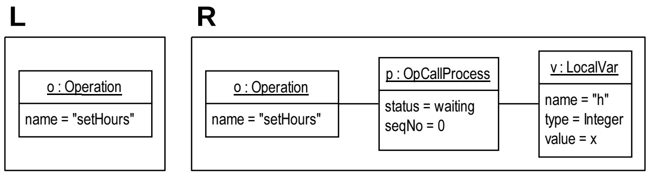

Figure 12. Rule for creating a use case process

With the rules described so far, we are now able to add processes to the system state in order to actually start a system run. Next we need rules that handle these processes, i.e., change the system state according to the semantics specified in the interaction diagrams. For every operation specified by an interaction diagram, we construct a set of rules. This holds for all operations no matter whether they belong to a class or to a use case. We call an operation, for which an interaction is given, a *user-defined* operation.

A user-defined operation calls several other operations. Which one and in which order is specified in an interaction diagram. An interaction dia-

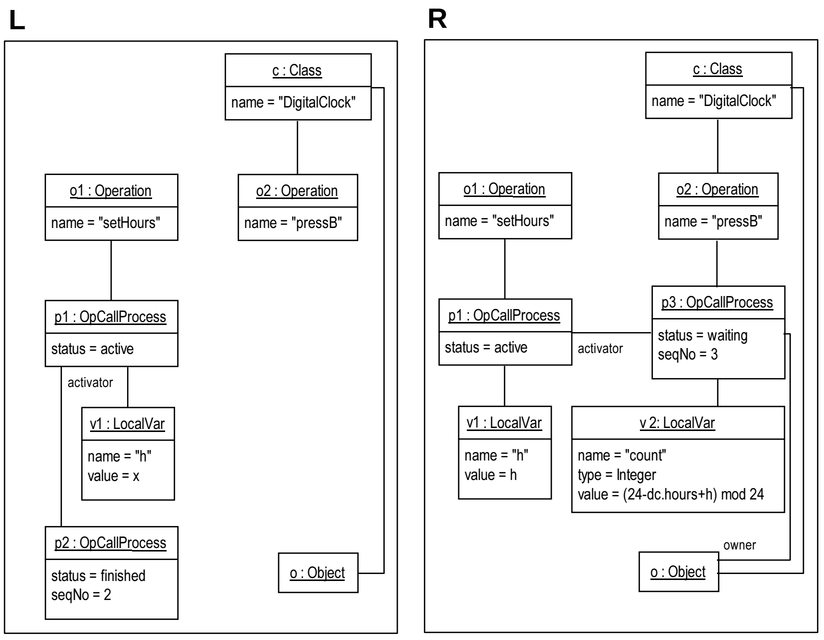

Figure 13. Rule for sending the message "3: pressB(h-dc.hours)"

gram contains messages sent between classifier roles in a specific order. Each message represents the call of either a user-defined operation (of a class) or it represents the call of a predefined functionality (like setting an attribute value). Every sent message corresponds to the creation of a process node, so we need a rule for each message of the interaction. We now examine the rule that "sends" the third message of the interaction diagram for setHours as depicted in Fig. 5. The rule is shown in Fig. 13.

The rule creates the process p3 (displayed in the middle of the right-hand side of the rule) belonging to the operation named pressB. A prerequisite for the rule to be applied is that a process of a setHours operation is active and that the process corresponding to the predecessor message is finished (as shown in the left-hand side of the rule by process p2). This is the reason why processes have a sequence number attribute seqNo. It is needed to refer to the process that has to be finished, in this case it is the one with the sequence number 2. In the right-hand side, the new process is connected to its activator, operation and owner object. In addition it has a newly created local variable compliant to the given parameter. The value of the local variable is an OCL expression that is evaluated when applying the rule. The predecessor does not occur in the right-hand side. It is removed from the system state because it is no longer needed.

If the attribute concurrency of the associated operation is guarded, an NAC ensures that no process of the same operation is already running on the owner. We do not show this NAC here.

4.2 Predefined Rules

Some messages do not call a user-defined operation but rather a predefined functionality. There are messages for creating an object of a specific class, destroying an object, connecting objects with a link of a given association, unlinking objects, setting an attribute value, setting a local variable value, and returning a result. Corresponding to these messages there are atomic processes that are not associated with an operation but instead with other information needed for the task. These atomic processes have already been shown in Fig. 10. The rules are constructed straight forward to realize the intended functionality.

Finally we have a rule for collecting garbage. This rule removes local variables that are no longer attached to a process node.

5 Implementation

Currently a prototype for the concepts discussed in this paper is being implemented. The goal of this prototype is to visualize the evolution of the system state. When provided with a model and an initial object diagram, the prototype automatically generates the graph transformation rules and the initial system state graph. A graphical user interface then permits the user to view the evolution of the system state step by step and to examine the current state by querying it using OCL.

For this reason the prototype must be able to perform graph transformations as well as evaluate OCL expressions. Instead of implementing a new tool for these purposes, we chose to combine two well established tools. The graph transformation part is done by AGG [16] and the evaluation of OCL expressions is performed by the USE tool [14].

The model and the initial object diagram are specified in USE-like syntax. The USE tool had to be extended in order to be able to read collaboration, use case, and statechart diagram definitions and to represent their features in its internal model representation. Furthermore the possibility to evaluate the oclInState operation had to be added to the OCL expression interpreter.

The core prototype comprises mainly two parts. The first part generates the graph transformation rules as well as the initial system state from the given model specification. This is achieved using the API of the second part of the prototype, which is visualized in Fig. 14. The main class of this part is the Grammar. This core class stores the generated graph transformation rules and the system state and deals with the actual rule application. As discussed earlier, a rule may contain OCL expressions as application conditions or as a means to calculate new attribute values. Since AGG is not able to handle OCL expressions, these rules cannot exactly be AGG rules. For this reason a system state rule has been developed, which extends the functionality of an AGG rule. The expressions calculating new attribute values are stored as constant string values in the corresponding attributes of the AGG nodes of the AGG rule, while the application conditions are stored directly in the system state rule.

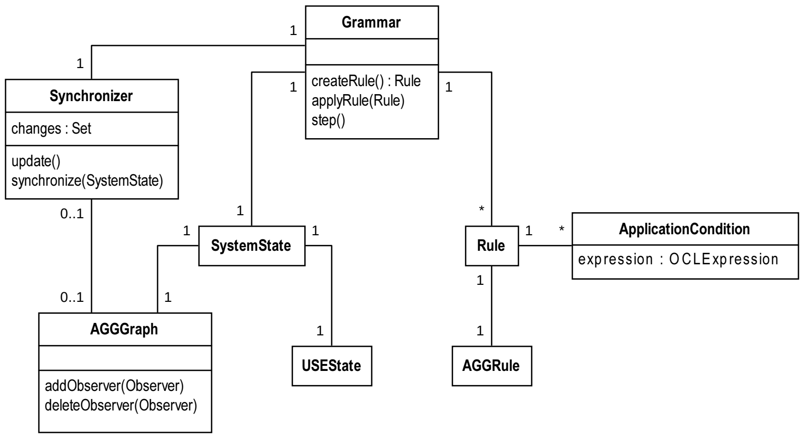

Figure 14. Simplified class diagram of the prototype

The system state grammar contains the system state. This system state combines the concepts of AGG and USE. A system state consists of an AGG working graph and a USE state. The AGG graph is necessary to be able to let AGG handle the pure graph rewriting, while the USE state is needed for the evaluation of OCL expressions. When applying a rule, the system state grammar has to ensure that the AGG graph and the USE state of the system state are synchronized. The application of a system state rule then works as follows.

If the rule has an application condition, the system state grammar uses the USE state and the USE expression evaluator to determine the value of given expressions depending on the match of the left-hand side of the rule to the system state. If it evaluates to true, the rule may be applied, otherwise another match is tested. If there is no further match, the application of the rule to the current system state is not possible. The actual application is then performed by AGG on the AGG graph of the system state. This comprises the correct handling of possible variables in the left-hand side of the rule. In order to keep the USE state of the system state synchronous to the AGG graph, the observer facility of AGG is used. An instance of class Synchronizer is registered as an observer of the observable AGG graph. Whenever this graph is changed, the synchronizer is informed via its update method. These changes are accumulated to change the USE state once AGG has completed the rule application. When the rule application is finished, the synchronizer is removed from the observer list of the AGG graph. At this stage, there may be unevaluated OCL expressions as string constants in attribute values of the working graph. They are now interpreted by USE and the values replace the corresponding constants. Then the system state grammar changes the USE state according to the previously collected changes of the AGG graph. Afterwards the USE state and the AGG graph represent the same system state and the next rule application may be calculated.

In order to provide the possibility to calculate the next step of a system state evolution, the system state grammar permits to apply any rule in the set of system state rules. This is done by randomly choosing one of the rules until an applicable one is found. Instead the user may also select a process to be executed. Furthermore the user may choose a new use case. In this case optional parameters have to be provided by the user. Note that a use case rule is always applicable and that such a use case rule needs to be selected to actually start a system state evolution.

6 Conclusion and Future Work

We have presented an integrated semantics for UML based on the translation of a given UML model into a graph transformation system. To demonstrate our approach an example model comprising several UML diagrams has been introduced. Next we have described our idea of a system state by means of a metamodel followed by a discussion of the translation of a given model into model-depending and predefined graph transformation rules by example. Finally the basic concepts of the prototypic software implementing this approach have been addressed. The prototype translates a given UML model into a graph transformation system and allows to monitor the evolution of the system state step by step.

The next goal is to complete the prototype implementation and to provide a convenient GUI for it. So eventually our approach can be evaluated in practice. As the approach and the tool areit is suitable for early stages of the software development process, it might become impractical when using large and very detailed models. In this case the aforementioned GUI should allow the user to choose different views on the system run, like e.g. hiding objects and their details that are of no interest in a certain situation.

An interesting topic would be the integration of further diagram types like activity diagrams into our approach. We will also investigate whether and how the diagrams already covered can be extended with yet missing UML features. Case studies will provide feedback on the practicability of the approach and tool. In particular, more insight is needed into the process of asserting properties of UML models on the basis of our approach, for instance, based on transformation invariants. In this way our approach will automatically benefit from future cognitions in the field of graph transformation.

References

[1] Boldsoft and Rational Software Corporation and IONA, *Response to the UML 2.0 OCL RfP (ad/2000-09-03)*, http://www.klasse.nl/ocl/ocl-subm.html.

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

[3] Cordes, B., K. Hölscher and H.-J. Kreowski, *UML interaction diagrams: Correct translation of sequence diagrams into collaboration diagrams*, in: M. Nagl, J. Pfaltz and B. Böhlen, editors, *AGTIVE'03 Proceedings*, Lecture Notes in Computer Science, 2004, to appear.

[4] Ehrig, H., G. Engels, H.-J. Kreowski and G. Rozenberg, editors, "Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools," World Scientific, Singapore, 1999.

[5] Ehrig, H., H.-J. Kreowski, U. Montanari and G. Rozenberg, editors, "Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3: Concurrency, Parallelism, and Distribution," World Scientific, Singapore, 1999.

[6] Engels, G., R. Heckel, J. M. Küster and L. Groenewegen, *Consistency-preserving model evolution through transformations*, in: J.-M. Jézéquel, H. Hussmann and S. Cook, editors, *UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings*, LNCS **2460** (2002), pp. 212–226.

[7] Fischer, T., J. Niere, L. Torunski and A. Zündorf, *Story diagrams: A new graph transformation language based on UML and Java*, in: H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, editors, *Proc. Theory and Application to Graph Transformations (TAGT'98), Paderborn, November, 1998*, LNCS **1764** (1998).

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

[9] Heckel, R. and S. Sauer, *Strengthening uml collaboration diagrams by state transformations*, in: H. Hussmann, 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*, LNCS **2029** (2001), pp. 109–123.

[10] Kuske, S., 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)*, LNCS **2335** (2002), pp. 11–28.

[11] Löwe, M., 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*, John Wiley, New York, 1993 pp. 185–199.

[12] OMG, "OMG Unified Modeling Language Specification, Version 1.5, March 2003," Object Management Group, Inc., Framingham, Mass., http://www.omg.org, 2003.

[13] Richters, M., "A Precise Approach to Validating UML Models and OCL Constraints," Ph.D. thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14 (2002).

[14] Richters, M., *A UML-based Specification Environment* (last revision 2001), http://www.db.informatik.uni-bremen.de/projects/USE.

[15] Rozenberg, G., editor, "Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations," World Scientific, Singapore, 1997.

[16] Rudolf, M. and G. Taentzer, *The Attributed Graph Grammar System AGG* (last revision 2003), http://tfs.cs.tu-berlin.de/agg.

[17] Sendall, S. and A. Strohmeier, *From use cases to system operation specifications*, in: A. Evans, S. Kent and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, LNCS **1939** (2000), pp. 1–15.

[18] Tsiolakis, A. 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, Berlin, March 2000*, 2000, Technical Report no. 2000/2, Technical University of Berlin.

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

[20] Ziemann, P. and K. Hölscher, *Example UML Model* (2004), http://www.tzi.de/~hwaters/example.pdf.