Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML

Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, Stefan Sauer

University of Paderborn, Dept. of Mathematics and Computer Science
D-33098 Paderborn, Germany
engels|corvette|reiko|sauer@uni-paderborn.de

Abstract. In this paper, dynamic meta modeling is proposed as a new approach to the operational semantics of behavioral UML diagrams. The dynamic meta model extends the well-known static meta model by a specification of the system's dynamics by means of collaboration diagrams. In this way, it is possible to define the behavior of UML diagrams within UML.
The conceptual idea is inherited from Plotkin's structured operational semantics (SOS) paradigm, a style of semantics specification for concurrent programming languages and process calculi: Collaboration diagrams are used as deduction rules to specify a goal-oriented interpreter for the language. The approach is exemplified using a fragment of UML statechart and object diagrams.
Formally, collaboration diagrams are interpreted as graph transformation rules. In this way, dynamic UML semantics can be both mathematically rigorous so as to enable formal specifications and proofs and, due to the use of UML notation, understandable without prior knowledge of heavy mathematic machinery. Thus, it can be used as a reference by tool developers, teachers, and advanced users.

Keywords: UML meta model, statechart diagrams, precise behavioral semantics, graph transformation

1 Introduction

The UML specification [20] defines the abstract syntax and static semantics of UML diagrams by means of (meta) class diagrams and OCL formulas. The dynamic (operational) semantics of behavioral diagrams is only described informally in natural language. However, when using UML models for communication between development teams, for project documentation, or as a contract between developers and customers, it is important that all partners agree on a common interpretation of the language. This requires a semantics specification which captures, in a precise way, both the structural and the dynamic features of the language.

Another fundamental requirement for the specification of a modeling language is that it should be readable (at least) by tool developers, teachers, and advanced users. Only in this way, a common understanding of the semantics of the language can be developed among its users.

Presently, most approaches to dynamic UML semantics focus on the implementation and simulation of models, or on automatic verification and reasoning. Reggio et al. [23], for example, use algebraic specification techniques to define the operational semantics of UML state machines. Lillius and Paltor [17] formalize UML state machines in PROMELA, the language of the SPIN model checker. Knapp uses temporal logic [15] for formalizing UML interactions. Övergaard [21] presents a formal meta modeling approach which extends static meta modeling with a specification of dynamics by means of a simple object-oriented programming language that is semantically based on the π-calculus. The formalisms used in the cited approaches provide established technologies for abstract reasoning, automatic verification, execution, or simulation of models, but they are not especially suited for explaining the semantics to non-experts.

In contrast, the technique of meta modeling has been successful, because it does not require familiarity with formal notations to read the semantics specification. Our approach to UML semantics extends the static meta model based on class diagrams [20] by a dynamic model which is specified using a simple form of UML collaboration diagrams. The basic intuition is that collaboration diagrams specify the operations of a goal-driven interpreter. For instance, in order to fire a transition in a statechart diagram, the interpreter has to make sure to be in the source state of the transition, and it might have to ask for the occurrence of a certain trigger event. This trigger event may in turn depend on the existence of a link mediating a method call, invoked by the firing of a transition in another statechart diagram, etc. Conceptually, this may be compared to the behavior of a Prolog interpreter trying to find a proof for a given goal.

Despite the graphical notation, the specification is mathematically rigorous since collaboration diagrams are given a formal interpretation based on graph transformation rules (see, e.g., [24, 6, 7] for a recent collection of surveys and [1] for an introductory text) within our approach. In particular, they can be considered as a special form of graphical operational semantics (GOS) rules [4], a generalization of Plotkin's structured operational semantics (SOS) paradigm for the definition of (textual) programming languages [22] towards graphs.

The paper is organized as follows: The approach to dynamic meta modeling is exemplified using an important fragment of UML statechart and object diagrams which is introduced along with a sample model in Sect. 2. In Sect. 3, we introduce the structural part of our meta model, a fragment of the standard meta model with meta classes extended by meta operations. The semantics specification in terms of collaboration diagrams is presented in Sect. 4, and in Sect. 5 it is shown how this specification can be used to compute the behavior of the sample model introduced in Sect. 2. Finally, in Sect. 6 we summarize and outline some future perspectives.

2 Statechart and Object Diagrams

Our approach to dynamic meta modeling shall be exemplified by the operational semantics of UML statechart and object diagrams. Statechart diagrams are used to specify the local behavior of objects (of a certain class) during their lifetime. Similarly to an event-condition-action rule, a transition consists of a triggering event, an activation condition, and a list of actions. Additionally, we regard the invocation of operations on an object as well as the calls to operations of other objects by the object under consideration as particularly relevant for this purpose. Therefore, we restrict our specification to transitions with call events and/or call actions. Conditions, other kinds of events and actions, composite and pseudo states, as well as more advanced structural concepts like inheritance and composition of classes are not considered.

The considered model extract refers to a problem of general importance, since the life cycle description of objects in a statechart diagram has to be related to the messaging mechanisms between interacting objects and the invocation of methods on such objects. A recent solution [3] suggests to model dynamic behavior by state machines and to view methods as private virtual objects to allow for concurrent execution by delegation. In contrast, we propose dynamic meta modeling as a basis for an integration of events, messages, and method invocation. In the following, we present an example that will allow us to demonstrate the application of our approach.

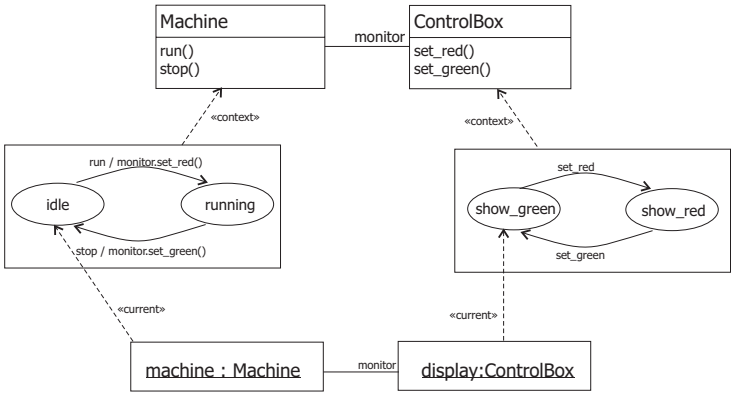

Fig. 1. A sample model (initial configuration)

Figure 1 shows a model consisting of two classes Machine and ControlBox related by an association stating that objects of class Machine may be monitored by objects of class ControlBox. In the Machine statechart diagram, transitions are labeled with combined event/action expressions like run/monitor.set_red(). That means, in order for the transition to fire, a call event for the operation run()

has to occur, and by firing the transition the method set_red() shall be called on the ControlBox object at the opposite end of the monitor link. As a result, the ControlBox object should change its state from show_green to show_red. No further actions are issued by the ControlBox statechart diagram. Notice that we do not model the implementation of operations. Therefore, the relevant interaction between objects (like switching the display by the machine) is described using call actions on the statechart level (rather than implementing it in the method run()).

The initial configuration of the system is given by an object diagram together with a specification of the control state of each object. In our example, machine is in state idle and display is in state show_green as shown in Fig. 1 by the stereotyped ≪current≫ relationships.

After presenting the static meta model and the firing rules of UML statechart diagrams in the next sections, we shall examine part of the life cycle of the objects introduced above.

3 Meta Classes and Meta Operations

In the UML semantics specification [20], the abstract syntax of statechart diagrams is specified by a meta class diagram. In order to define the structural model of an interpreter for this languages, this model has to be extended by state information, for example to represent the current control state of an object.

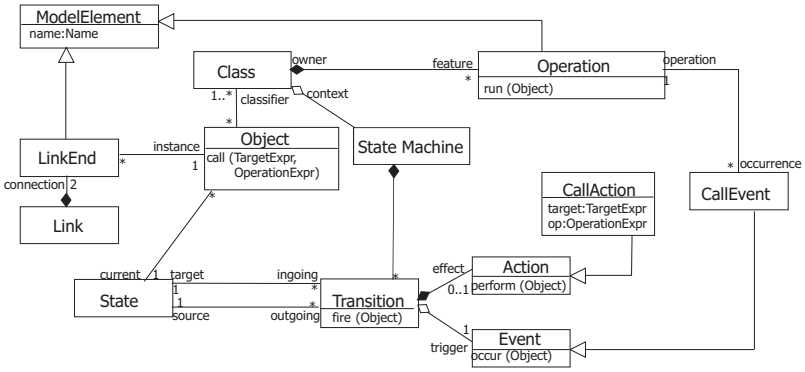

Fig. 2. Meta class diagram

Figure 2 shows the classes from the UML meta model that are relevant for the subclass of statechart diagrams we are considering (partly simplified by flattening the meta class hierarchy). A statechart diagram, represented by an instance of meta class StateMachine, controls the behavior of the objects of the class it is associated with. For this purpose, we extend the meta model by an association

current which designates the current control state of an object within the state diagram. States and transitions are represented by instances of the corresponding meta classes, and transitions are equipped with a trigger CallEvent (like run in our scenario) and an effect CallAction (like control.set_red()). A CallEvent carries a link to the local operation which is called. Unlike in the standard meta model, a CallAction is not directly associated with an operation, as this would result in static binding. Instead, an attribute OperationExpr is provided.

The state space of the diagrammatic language consists of all instance graphs conforming to the meta class diagram. Each instance graph represents the state of an interpreter given by the "programs" (e.g., statechart diagrams) to be executed, the problem domain objects with their respective data states (given, e.g., by the values of attributes and links), and their control states.

The relation between class and instance diagrams can be formally captured by the concept of *type* and *instance graphs* [5].[1] Given a type graph TG, representing a class diagram, a *TG-typed instance graph* 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. For example, the instance graph of the meta class diagram in Fig. 2 that represents the abstract syntax of the model in Fig. 1 is shown in Fig. 3.

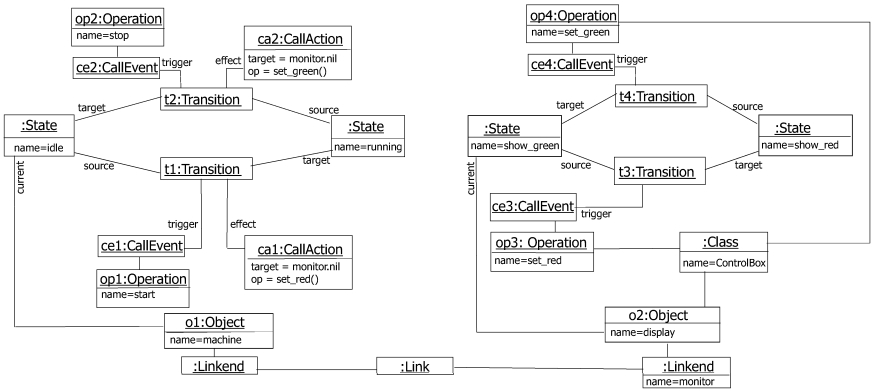

Fig. 3. Abstract syntax of sample model

The class diagram in Fig. 2 does not only contain meta classes and associations, but also *meta operations* like perform(Object) of class Action. They are the operations of our interpreter for statechart diagrams. Given the type graph

[1] 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. 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.

TG representing the structural part of the class diagram, the meta operations form a family of sets $M = (MOP_w)_{w \in TG_V^+}$ indexed by non-empty sequences $w = v_1 \dots v_n$ of parameter class names $v_i \in TG_V$. By convention, the first parameter v_1 of each meta operation represents the class to which the operation belongs (thus there has to be at least one argument type). For example, the meta operation perform(Object) of class Action is formally represented as $\mathsf{perform} \in MOP_{\mathsf{Action,Object}}$.

After having described the abstract syntax of our model in terms of meta classes and meta operations, the implementation of the meta operations shall be specified using collaboration diagrams in the next section.

4 Meta Modeling with Collaboration Diagrams

The static meta model of the UML defines the abstract syntax of the language by means of meta class diagrams. Seen as a system specification, these class diagrams represent the structural model of an UML editor or interpreter. In this section, we shall extend this analogy to the dynamic part of a model, i.e., we are going to specify the dynamics of an interpreter for statechart and object diagrams. Interaction diagrams and, in particular, collaboration diagrams are designed to specify object interaction, creation, and deletion in a system model. Dynamic meta modeling applies the same language concepts to the meta model level to specify the interaction and dynamics of model elements of the UML.

The specification is based on the intuition of an interpreter which has to demonstrate the existence of a certain behavior in the model. Guided by a recursive set of rules stating the conditions for the execution of a certain meta operation, the interpreter works its way from a goal (e.g., the firing of a transition) towards its assumptions (e.g., the occurrence of a trigger event). The behavioral rules are specified by collaboration diagrams consisting of two compartments. The head of the diagram contains the meta operation which is specified by the diagram. The body specifies the assumptions for the execution of the meta operation, its effect on the object configuration, and other meta operations required.

For example, the conditions for a transition to fire and its effect on the configuration are specified in the collaboration diagram of Fig. 4: An object o may fire a transition if that object is in the corresponding source state, the (call) event triggering the transition occurs, and the operation associated with this call event is invoked by the meta operation run(o). In this case, the object o changes to the target state of the transition, which is modeled by the deletion and re-creation of the current link.

Thus, in order to be able to continue, the interpreter looks for a call event triggering the transition. This call event can be raised if the associated operation is called on the object o as specified in Fig. 5 using the meta operation call. The signature of this meta operation of meta class Object contains two parameters: The first one holds a path expression to direct the call to its target object (it equals nil when the target object is reached), and the second one specifies the

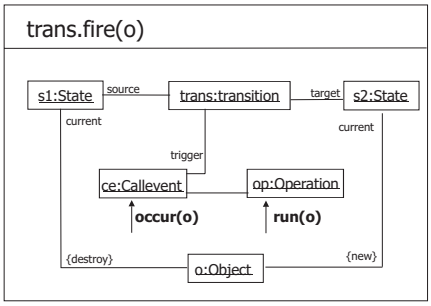

Fig. 4. The firing of a transition by an object

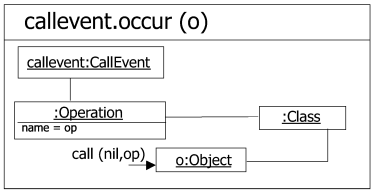

Fig. 5. Issuing a CallEvent

name of the operation to be called (and possibly further parameters). The name of the operation op has to match the operation expression transmitted by call.

Note that this does not guarantee the execution of the body of the called operation. In fact, no rule for meta operation run of meta class Operation is provided. The specification of the structure and dynamics of method implementations is the objective of *action semantics* as described by the corresponding request for proposals [18] by the Object Management Group. So far, UML provides only "uninterpreted strings" to capture the implementation of methods. We believe that our approach is extensible towards a dynamic semantics of actions once this is precisely defined.

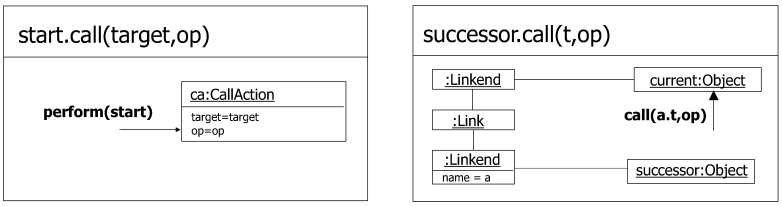

Fig. 6. Evaluating the target expression

An operation call like o.call(nil, op) in Fig. 5 originates from a call action which specifies by means of a path expression the target of the call. Thus, in order to find out whether a call is pending for a given object o, our interpreter has to check two alternatives: Either a call action is performed on o directly with target = nil, or there is a call at a nearby object with a target expression pointing towards o. These two cases are specified by the two collaboration diagrams for meta operation call in Fig. 6. The left diagram specifies the invocation of the meta operation by a CallAction on an object start. (The object is not depicted since it is given by the parameter of the premise.) Notice that the values of the meta attributes target and op have to match the parameters of meta operation call.

If the meta operation is not directly invoked by a call action, an iterative search is triggered as specified by the right diagram: To invoke the meta operation call(t,op) on an object successor which is connected to object current via a link, whose link end named a is attached to the successor object, the meta operation call(a.t,op) has to be invoked on current with the identical operation parameter op and the extended path expression a.t. (We assume target to be in a Java-like path syntax where the names of the links to be followed form a dot-separated list.)

Notice, that the right rule in Fig. 6 is potentially non-deterministic: In a state where the successor object has more than one incoming a link, different instantiations for the current object are possible. In this case, the link to be followed would be chosen non-deterministically.

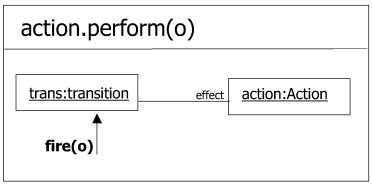

Fig. 7. The performing of an action by an object

Figure 7 presents the rule for performing an action. In our scenario this should be a CallAction initiating a call to another object, but the rule is also applicable to other kinds of actions. An action is the (optional) effect of firing a transition, i.e., the invocation of meta operation perform of meta class Action depends on the firing of the associated transition. Thus, the rule in Fig. 4 has to be applied again in order to derive the firing of the transition at the calling object.

As already mentioned in the introduction, this goal-oriented style of semantics specification is conceptually related to the proof search of a Prolog interpreter. This intuition is made precise by the paradigm of graphical operational semantics (GOS) [4], a graph-based generalization of the structured operational semantics (SOS) paradigm [22], for the specification of diagram languages. In

the GOS approach, deduction rules on graph transformations are introduced in order to to formalize the derivation of the behavior of models from a set of meta-level diagrams, which is implicitly present in this section. In the next section, we describe a simplified form of this approach especially tailored for collaboration diagrams.

5 Computing with Collaboration Diagrams

In the previous section, collaboration diagrams have been used to specify the firing rules of statechart transitions and the transmission of calls between objects. Now, concrete computations shall be modeled as collaboration diagrams on the instance level. This allows us to represent changes to the object structure together with the operations causing these changes. Moreover, even incomplete computations can be modeled, where some of the method calls are still unresolved. This is important if we want to give semantics to incomplete models like the one in Sect. 2 which requires external activation in order to produce any activity.

The transition from semantic rules to computations is based on a formal interpretation of collaboration diagrams as graph transformation rules. A rule representing the collaboration diagram for operation trans.fire(o) in Fig. 4 is shown in Fig. 8. It consists of two graphs L and R representing, respectively, the pre- and the post-condition of the operation. In general, both L and R are instances of the type graph TG representing the class diagram, and both are subgraphs of a common graph C that we may think of as the object graph of the collaboration diagram. Then, the pre-condition L contains all objects and links which have to be present before the operation, i.e., all elements of C except for those marked as {new} or {transient}. Analogously, the post-condition contains all elements of C not marked as {transient} or {destroy}. In the example of Fig. 8, graph C is just the union $L \cup R$ since there are no transient objects in the diagram of Fig. 4.

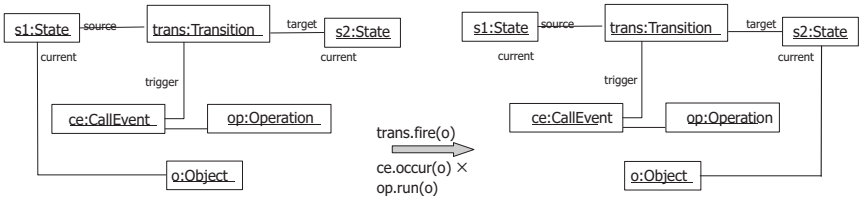

Fig. 8. Collaboration diagram as a labeled graph transformation rule

Besides structural modifications, the collaboration diagram describes calls to meta operations ce.occur(o) and op.run(o), and it is labeled by the operation trans.fire(o), the implementation of which it specifies. This information is

recorded in the rule-based presentation in Fig. 8 by means of additional labels above and below the arrow. Abstractly, a collaboration diagram is denoted as

$$C : L \xrightarrow[b]{a} R$$

where C is the object graph of the diagram, L and R are the pre- and post-conditions, a is the label representing the operation specified by the diagram, and b represents the sequential and/or concurrent composition of operations referred to (that is, called) within in the diagram. The expression ce.occur(o) $\times$ op.run(o) in Fig. 8, for example, represents the concurrent invocation of two operations.

We shall use the rule-based interpretation of collaboration diagrams in order to derive the behavior of the sample model introduced in Sect. 2. The idea is to combine the specification-level diagrams by means of two operators of *sequential composition* and *method invocation*. The sequential composition of two diagrams

$$C_1 : L_1 \xrightarrow[b_1]{a_1} R_1 \text{ and } C_2 : L_2 \xrightarrow[b_2]{a_2} R_2$$

is defined if the post-condition R_1 of the first equals the pre-condition L_2 of the second. The composed diagram is given by

$$C_1 \cup_{L_2=R_1} C_2 : L_1 \xrightarrow[b_1\,;\,b_2]{a_1\,;\,a_2} R_2$$

where $C_1 \cup_{L_2=R_1} C_2$ denotes the disjoint union of the graphs C_1 and C_2, sharing only $L_2 = R_1$. The second operator on diagrams models the invocation of a method from within the implementation of another method. This is realized by substituting the method call by the implementation of the called method, thus diminishing the hierarchy of method calls. Assume two rules

$$C : L \xrightarrow[b[c]]{a} R \text{ and } C' : L' \xrightarrow[d]{c} R'$$

where the call expression $b[c]$ of the first rule contains a reference to the operation c specified by the second rule. (In the rule of Fig. 8, $b[c]$ corresponds to ce.occur(o) $\times$ op.run(o), and c could be instantiated with either ce.occur(o) or op.run(o).) Then, the composed rule is given by

$$C \cup_c C' : L \cup_c L' \xrightarrow[b[d]]{a} R \cup_c R'.$$

The call to c is substituted by the expression d specifying the methods called within c. By $C \cup_c C'$ we denote the union of graphs C and C' sharing the *self* and parameter objects of the operation c.[2] In the same way, the pre- and post-conditions of the called operation are imported inside the calling operation.

[2] Notice that, in order to ensure that the resulting diagram is consistent with the cardinality constraints of the meta class diagram, it might be necessary to identify further elements of C and C' with each other (besides the ones identified by c). For instance, when identifying two transitions, we also have to identify the corresponding source and target states. Formally, this effect is achieved by defining the union as a pushout construction in a restricted category of graphs (see, e.g., [16]).

In Fig. 9 it is outlined how these two composition operators are used to build a collaboration diagram representing a possible run of our sample model. The given diagrams are depicted in iconized form with sequential composition and invocation as horizontal and vertical juxtaposition, respectively. This presentation is inspired by the *tile model* [12], a generalization of the SOS paradigm [22] towards open (e.g., incomplete) systems. In fact, in our example, such a semantics is required since the model in Fig. 1 is incomplete, i.e., it does not specify the source of the call events run and stop needed in order to trigger the machine's transitions.

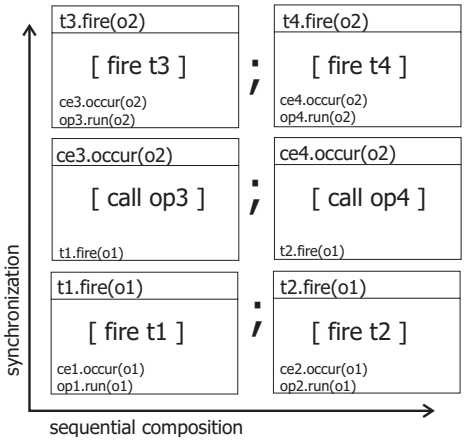

Fig. 9. Composing a run of the sample model

Figure 10 shows an expanded version of the iconized diagram *[fire t3]* in the top left of Fig. 9. It originates from an application of the operation trans.fire(o) in the context of an additional transition.[3] The diagrams *[fire t4]* to the right of *[fire t3]* as well as *[fire t1]* and *[fire t2]* in the bottom are expanded analogously.

Figure 11 details the icon labeled *[call op3]*. It shows the invocation of several operations realizing the navigation of the method call along the monitor link as specified by the target expression, and the issuing of the call event. A similar diagram could be drawn for *[call op4]*.

Finally, in Fig. 12 the composite computation is shown covering the complete scenario depicted in Fig. 1. It can be derived from the components in Fig. 9 in two different ways: by first synchronizing the single transitions (vertical dimension)

[3] In general, contextualization of rules has to be specified explicitly in our model (in this we follow the philosophy of the SOS and the tile framework [22, 12]). In the present specification, however, we can safely allow to add any context but for the current links which ensure the coordinated behavior of the different statechart diagrams.

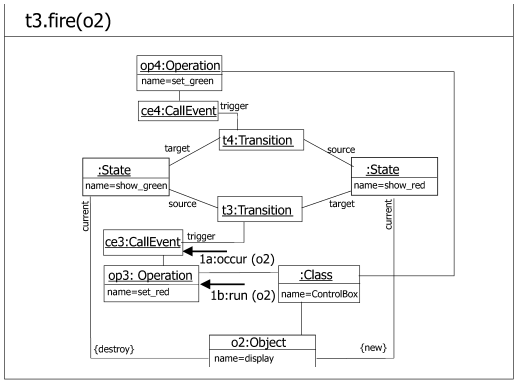

Fig. 10. Operation trans.fire(o) in context

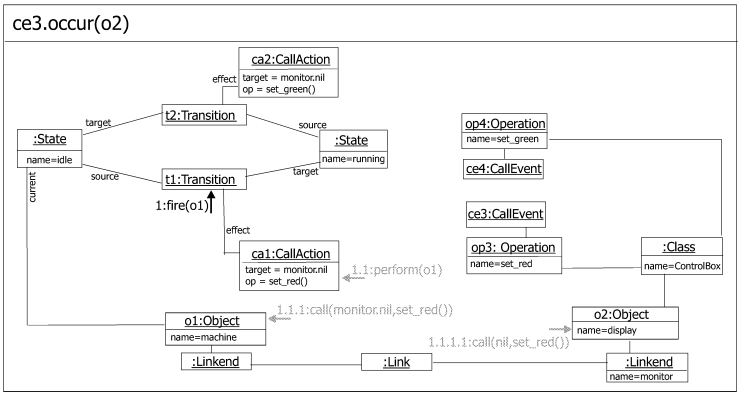

Fig. 11. Navigation of the method call

and then sequentially composing the two steps (horizontal dimension), or first building local two-step sequences (horizontal dimension) and then synchronizing them (vertical dimension).

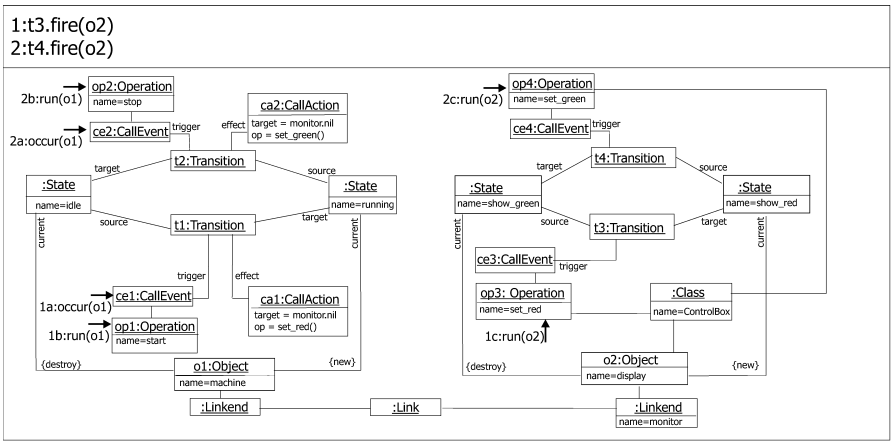

Fig. 12. Composite rule for the scenario in Fig. 1

6 Conclusion

In this paper, we have proposed the use of collaboration diagrams formalized as graph transformation rules for specifying the operational semantics of diagram languages. The concepts have been exemplified by a fragment of a dynamic meta model for UML statechart and object diagrams.

The fragment should be extended to cover a semantically complete kernel of the language which can be used to define more specific, derived modeling concepts. This approach is advocated by the *pUML* group (see e.g., [9]). Concrete examples how to define such a mapping of concepts include the flattening of statecharts by means of graph transformation rules [13] and the simplification of class diagrams [14] by implementing inheritance in terms of associations.

Our experience with specifying a small fragment of UML shows that tool support is required for testing and animating the specification. While the implementation of flat collaboration diagrams is reasonably well understood (see, e.g., [8, 10]), the animation of the results of an execution on the level of concrete syntax is still under investigation. It requires a well-defined mapping between the concrete and the abstract syntax of the modeling language. One possible solution is to complement the graph representing the abstract syntax by a *spatial relationship graph*, and to realize the mapping by a graphical parser specified by a graph grammar [2].

A related problem is the integration of model execution and animation in existing UML tools. Rather than hard-coding the semantics into the tools, our approach provides the opportunity to allow for *user-defined semantics*, e.g., in the context of domain-specific profiles. Such a profile, which extends the UML standard by stereotypes, tagged values, and constraints [19], could also be used to implement the extensions to the static meta model that are necessary in order to define the operational semantics (e.g., the current links specifying the control states of objects could be realized as tagged values).

On the more theoretical side, the connection of dynamic meta modeling with proof-oriented semantics following the SOS paradigm allows the transfer of concepts of the theory of concurrent languages, like bisimulation, action refinement, type systems, etc. Like in the GOS approach [4], the theory of graph transformation can provide the necessary formal technology for transferring these concepts from textual to diagram languages.

References

1. M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34:1–54, 1999.
2. R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of graph transformation to visual languages. In Ehrig et al. [6], pages 105–180.
3. R. Breu and R. Grosu. Relating events, messages, and methods of multiple threaded objects. *JOOP*, pages 8–14, January 2000.
4. A. Corradini, R. Heckel, and U. Montanari. Graphical operational semantics. In A. Corradini and R. Heckel, editors, *Proc. ICALP2000 Workshop on Graph Transformation and Visual Modelling Techniques, Geneva, Switzerland*, Geneva, July 2000. Carleton Scientific.
5. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.
6. 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.
7. 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.
8. G. Engels, R. Hücking, St. Sauer, and A. Wagner. UML collaboration diagrams and their transformation to Java. In R. France and B. Rumpe, editors, *Proc. UML'99 Int. Conference, Fort Collins, CO, USA*, volume 1723 of *LNCS*, pages 473–488. Springer Verlag, October 1999.
9. A. Evans and S. Kent. Core meta modelling semantics of UML: The pUML approach. In France and Rumpe [11], pages 140–155.
10. T. Fischer, 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. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn, November 1998*, volume 1764 of *LNCS*. Springer Verlag, 2000.

11. R. France and B. Rumpe, editors. *Proc. UML'99 – Beyond the Standard*, volume 1723 of *LNCS*. Springer Verlag, 1999.
12. F. Gadducci and U. Montanari. The tile model. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1999.
13. M. Gogolla and F. Parisi-Presicce. State diagrams in UML – a formal semantics using graph transformation. In *ICSE'98 Workshop on Precise Semantics of Modelling Techniques*, 1998. Tech. Rep. TUM-I9803, TU München.
14. M. Gogolla and M. Richters. Equivalence rules for UML class diagrams. In P.-A. Muller and J. Bezivin, editors, *Proc. UML'98 Workshop*, pages 86–97. Universite de Haute-Alsace, Mulhouse, 1998.
15. A. Knapp. A formal semantics of UML interactions. In France and Rumpe [11], pages 116–130.
16. M. Korff. Single pushout transformations of equationally defined graph structures with applications to actor systems. In *Proc. Graph Grammar Workshop, Dagstuhl, 1993*, volume 776 of *LNCS*, pages 234–247. Springer Verlag, 1994.
17. J. Lillius and I. Paltor. Formalising UML state machines for model checking. In France and Rumpe [11], pages 430–445.
18. Object Management Group. Action semantics for the UML, November 1998. `http://www.omg.org/pub/docs/ad/98-11-01.pdf`.
19. Object Management Group. Analysis and design platform task force – white paper on the profile mechanism, April 1999. `http://www.omg.org/pub/docs/ad/99-04-07.pdf`.
20. Object Management Group. UML specification version 1.3, June 1999. `http://www.omg.org`.
21. G. Övergaard. Formal specification of object-oriented meta-modelling. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE'00), Berlin, Germany*, number 1783 in LNCS, pages 193–207. Springer Verlag, March/ April 2000.
22. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981.
23. G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML active classes and associated state machines – a lightweight formal approach. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE'00), Berlin, Germany*, number 1783 in LNCS, pages 127–146. Springer Verlag, March/ April 2000.
24. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.