Modeling Behavior with Interaction Diagrams in a UML and OCL Tool

Martin Gogolla, Lars Hamann, Frank Hilken, Matthias Sedlmeier

Database Systems Group, University of Bremen, Germany
{gogolla|lhamann|fhilken|ms}@informatik.uni-bremen.de

Abstract. This paper discusses system modeling with UML behavior diagrams. We consider statecharts and both kinds of interaction diagrams, i.e., sequence and communication diagrams. We present new implementation features in a UML and OCL modeling tool: (1) Sequence diagram lifelines are extended with states from statecharts, and (2) communication diagrams are introduced as an alternative to sequence diagrams. We assess the introduced features and propose selection mechanisms which should be available in both kinds of interaction diagrams. We emphasize the role that OCL can play for such selection mechanisms.
Keywords. UML, OCL, Model behavior, Statechart diagram, Interaction diagram, Sequence Diagram, Communication Diagram, Model validation, Diagram view.

1 Introduction

In the last years the Unified Modeling Language (UML) has become a de-facto standard for the graphical design of IT systems. UML [19, 21] comprises language features for structural and behavioral modeling. The textual Object Constraint Language (OCL) as part of UML adds precision in form of class invariants for restricting structural aspects and pre- and postconditions for constraining behavioral ones, among other uses of OCL [20, 23] within UML.

This contribution puts emphasis on UML interaction diagrams which are syntactically presented in form of sequence and communication diagrams. Interactions describe sequences of messages exchanged among parts of a system. We use interactions for the analysis of a system which has been described structurally with a class diagram including class invariants and behaviorally with operation pre- and postconditions, operation implementations, and statecharts. In general, behavioral diagrams have become more important in the modeling of systems. The specification of interactions using the respective behavior diagrams is more understandable, which is one of the goals of the UML. In addition, the specification of actions is more intuitive using diagrams instead of textual OCL pre- and postconditions, which is widely used for, e.g., business services. We introduce new features for interactions in a UML tool and discuss how the two interaction diagrams could be handled in a uniform way.

Our group is developing the UML and OCL tool USE (UML-based Specification Environment) since about 15 years. USE [7, 10] originally started as a kind of OCL interpreter with class, object and sequence diagrams available in the tool from the beginning. Other behavioral diagrams have been added over the last years, namely statechart diagrams in form of protocol state machines and most recently communication diagrams. USE claims to be useful for validation and verification of UML and OCL models. USE has been employed successfully in national and international projects (see, for example, [1] and [6] among other projects).

The rest of this paper is structured as follows. Section 2 introduces a running example. After having set with the example the context of our work, we discuss in Sect. 3 some general issues concerning behavioral modeling: 'abstraction', 'best practices', and 'tool support'. Section 4 explains in more details how our system USE contributes to system validation and verification. Section 5 shows the UML metamodel for interactions and sets the context for the interaction diagram implementation within USE. Section 6 presents new features in sequence diagrams, and Sect. 7 discusses established and new features in communication diagrams. In Sect. 8 a direct comparison between the two interaction diagrams is shown. Section 9 proposes systematic selection mechanisms that could be available in both interaction diagrams. Section 10 compares our approach to related papers. The contribution is closed in Sect. 11 with concluding remarks and future work.

2 Running Example

This section explains a running example which is used throughout the paper. In Fig. 1, a small, abstract version of Toll Collect[1] is shown. Toll Collect is a tolling system for trucks on German motorways. In the figure, the following USE features are employed: (a) a class diagram with two classes, (b) two statecharts (two protocol state machines) for each of the classes, (c) one object diagram, (d) one list of commands representing a scenario (test case), and the evaluation of (e) the class invariants and (f) a stated OCL query expression in the system state that is reached by executing the command list. The reached system state is characterized by the object diagram.

The class diagram consists of a part responsible for building up the motorway connections (basically `Point`, `Connection`, `northConnect(Point)`, `southConnect(Point))` and a part for managing trucks and journeys (basically `Truck`, `Current`, `enter(Point)`, `move(Point)`, `pay(Integer)`). The model includes three OCL class invariants (restricting system structure) and a number of OCL operation contracts in form of pre- and postconditions (restricting system behavior). Apart from the above used standard UML descriptions, the operations are implemented in a Simple OCL-like Imperative programming Language (SOIL). An example for an operation contract and an operation imple-

[1] www.toll-collect.de/en/home.html

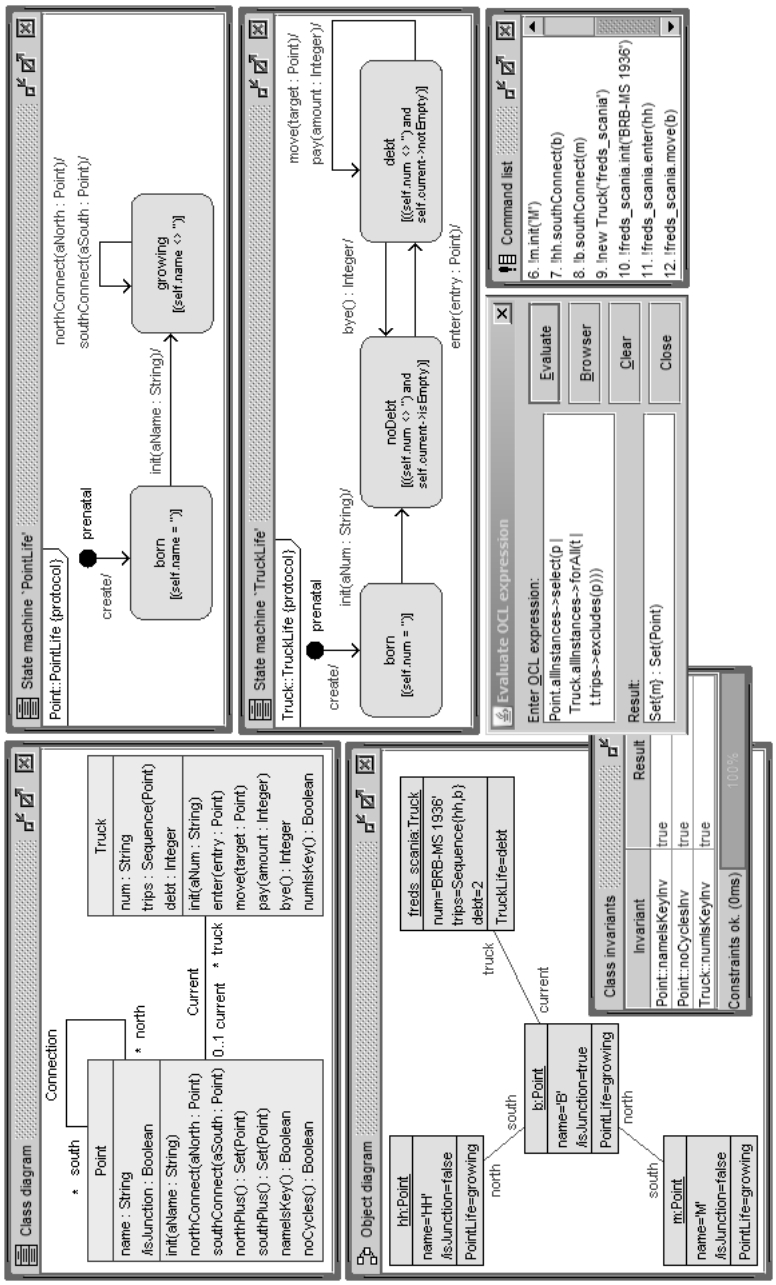


Fig. 1. Example model Toll Collect.

```
Truck::move(target:Point)
  begin self.trips:=self.trips->including(target);
        self.debt:=self.debt+1;
        delete (self,self.current) from Current;
        insert (self,target) into Current;
  end
pre currentExists:
  self.current->notEmpty
pre targetReachable:
  self.current.north->union(self.current.south)->includes(target)
post debtIncreased:
  self.debt@pre+1=self.debt
post tripsUpdated:
  self.trips@pre->including(target)=self.trips
post currentAssigned:
  target=self.current
post allTruckInvs:
  numIsKey()
```

Fig. 2. Example of operation implementation and pre- and postconditions.

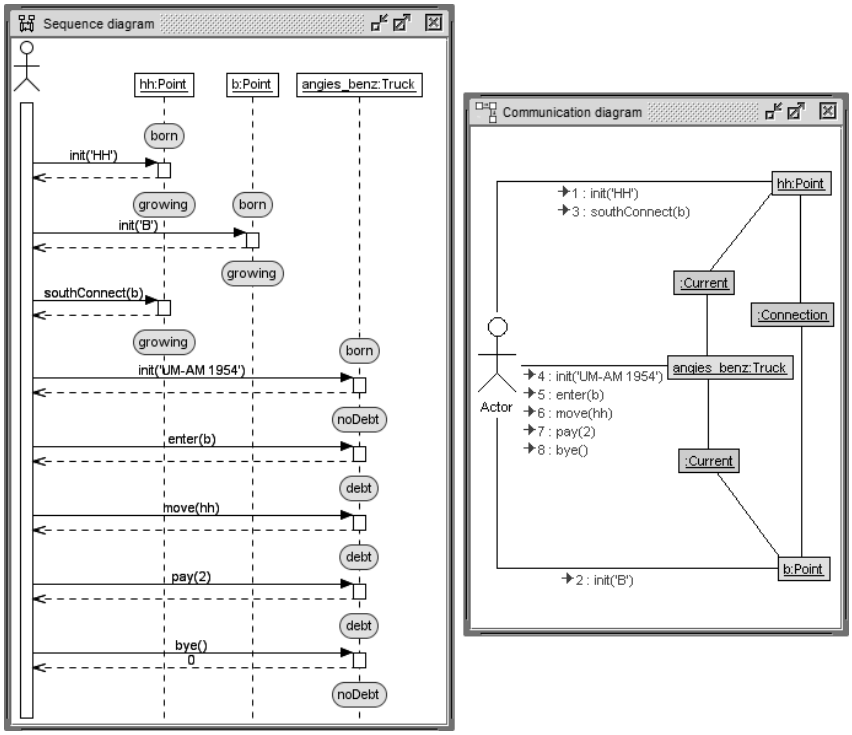


Fig. 3. Sequence diagram with statechart states on lifelines (some details suppressed) and equivalent communication diagram.

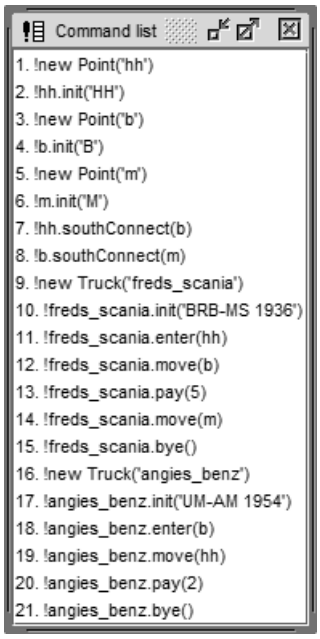


Fig. 4. Command list for used interaction diagrams.

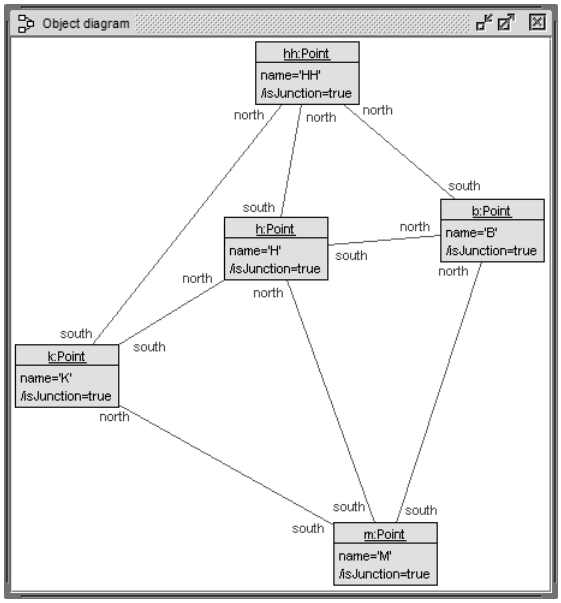


Fig. 5. Example for motorway connections.

mentation in SOIL [2] is shown in Fig. 2. Figure 3 displays a shortened variation of the scenario that the paper will discuss in detail in form of a sequence diagram and an equivalent communication diagram.

In Fig. 4, we show a longer command list where the single commands either generate objects with a specified object identity or call operations on generated objects. This command list and the commands determined by the respective operation implementation in SOIL are used in the following as the basis for the discussed interaction diagrams. This command list represents one test case, and this test case shows the *consistency* of the operation contracts in the sense that at least one scenario is possible where all operations are called (and thus *all pre- and postconditions* are valid) and *all invariants* are valid at times when no operation is active. The considered motorway connections are a toy example with the largest German towns Hamburg (`hh`), Berlin (`b`), and Munich (`m`). A slightly larger motorway example allowing to travel between western and eastern points as well is shown in Fig. 5. The complete USE model is given in the Appendix.

3 General Behavioral Modeling Issues: Abstraction, Best Practices, Tool Support

Before we go into the details of our approach we want to discuss crucial questions between our work and general issues in behavioral modeling: To what extent does our approach support behavioral modeling *abstraction* mechanisms? What is the relationship between our proposal and established *best practices* in behavioral modeling? How is our work supported by *tools*?

Abstraction: The motivation for modeling and the relationship to abstraction has been formulated to the point in [22] (and other works by the same author): *Why do engineers build models? (a) To understand problems and solutions, (b) to communicate model and design intent, (c) to predict interesting characteristics of the system under study, and (d) to specify the implementation of the system under study. Building models is realized by selecting statements through abstraction, i.e., reduction of information preserving properties relative to a given set of concerns.*

In our view structural and behavioral modeling must go hand in hand. As our background is database and information system modeling, we typically start with structural modeling and later involve behavioral aspects. Other IT disciplines as, for example, embedded systems may prefer to start with behavioral issues and continue with structural ones. In our view, behavioral aspects are inherently more complex than structural issues because in information systems the behavioral descriptions must be aware of and respect the structural requirements. Thus finding good abstraction techniques that *reduce information* are even more relevant for behavioral modeling.

As said already before, we use UML interactions for the analysis of a system which has been described structurally with a class diagram including class

invariants and behaviorally with operation pre- and postconditions, operation implementations, and statecharts. One focus here is on UML interaction diagrams in form of communication diagrams. Communication diagram are able to present all details of a behavioral scenario and bear the danger to overwhelm the modeler with too many messages which are the basic cornerstones of a scenario. Thus in particular for communication diagrams proper and adequate abstraction mechanisms are strongly needed. This demand leads in our approach to a proposal for allowing views on interaction diagrams that take into account message number intervals, message depth, and message kind abstraction mechanisms in order to show that part of a scenario that the modeler regards as important.

Best practices: UML sequence and communication diagrams are employed for showing interactions, i.e., message exchanges between objects (or object roles) in order to perform a task. *Both sequence and communication diagrams show interactions, but they emphasize different aspects. A sequence diagram shows time sequence as a geometric dimension, but the relationship among [object] roles are implicit. A communication diagram shows the relationships among [object] roles geometrically and relates messages to the connectors, but time sequences are less clear because they are implied by the sequence numbers. Each diagram should be used when its main aspect is the focus of attention* (quoted from [21]). If one wants to capture the difference along the slogan *Time vs Space*, one would classify the sequence diagram into the *Time* dimension and the communication diagram into the *Space* dimension.

However, there is only little methodological help on the question when to use which diagram. Our observation is that sequence diagrams are more frequently used than communication diagrams. It seems that sequence diagrams can be used intuitively easier due to explicitly displayed message order. The message order must be mentally retrieved in communication diagrams. However, as said before, communication diagrams show the relationship between objects which is neglected in sequence diagrams.

Tool support: Both sequence and communication diagrams are supported by UML tools. However, a general common view mechanism on the underlying interactions is not explicitly stated in UML. This leads to different features for interactions diagrams in different tools.

Our proposal here is to offer the same view mechanisms in both interaction diagrams. The motivation for an (as far as possible) uniform treatment of sequence diagrams and communication diagrams comes from the fact that both diagram forms treat the same model elements: interactions, i.e., objects and messages between them. For example, if one starts from a complex interaction in form of a sequence diagram and one selects a subset of the involved objects for viewing, then it should be possible to do the same selection in the corresponding communication diagram. The same holds if the selection is made for messages. A conversion between both diagram forms is in principle possible because of identical underlying elements (objects and messages) and because of the fact that the geometrical ordering in the sequence diagram

has its equivalent in the numerical ordering in the communication diagram. However the relationships between objects present in the communication diagram do not have an equivalent in the sequence diagram and thus cannot be represented. With respect to the underlying static structure (the class diagram) both interaction diagrams use the same elements arising from the class diagram, basically commands for the creation and deletion of objects and links, for the manipulation of attributes and for operation calls.
Interaction diagrams can be looked at from different angles. One can view interactions in both sequence and communication diagrams along the object or along the message dimension. Furthermore, apart from interactively selecting relevant parts in a scenario, we discuss how to employ OCL for systematically accessing objects and messages.
The discussed features are implemented in our tool USE. Sequence diagrams have been present in USE from the very beginning, and only later communication diagrams were added. Integrated views on both kinds of interaction diagrams with common features are currently under development. The aim of the newly added view features is to better support new abstraction mechanisms for behavioral modeling, in particular in connection with communication diagrams that are only poorly supported in present UML tools as far as voluminous scenarios are concerned.

4 Validation and Verification with USE

OCL can be employed in USE for various tasks: in class diagrams for (a) class invariants, (b) operation contracts, (c) attribute and association derivation rules, and (d) attribute initializations; in protocol state machines for (e) state invariants and (f) transition pre- and postconditions; furthermore for (g) ad-hoc OCL queries in object diagrams, and for (h) expressions within SOIL. In USE, class diagrams and protocol machines enriched by invariants, operation contracts, statechart constraints and SOIL operation implementations determine system structure and behavior. Sequence and communication diagrams are employed in USE for visualizing and analyzing specified test cases in form of scenarios. Interaction diagrams are not used for restricting system behavior, but to document, analyze, and understand the interactions. These diagrams are built after a complete model including the SOIL operation implementation has been constructed.

The overall aim of USE is to support development by reasoning about the model through (a) validation, i.e., checking informal expectations against formally given properties, for example, by stating OCL queries against a reached system state (object diagram) and (b) verification, i.e., checking formal properties of the model, for example by considering model consistency or the independence of invariants as in [7]. That contribution also shows how USE supports making deductions from the stated model on the basis of a finite search space of possible system states (object diagrams). Such checks are realized in form of positive and negative scenarios which can be thought of as being test cases for the system under consideration. Thus USE supports the development of tests.

In OCL operation contracts as well as pre- and postconditions can be general OCL formulas. In postconditions, one can refer with `@pre` to attribute and association end values at precondition time. Postconditions can state general requirements and are not restricted to the specification of changes to attribute and association end values. Thus the actual changes made by the operation are described in SOIL and are checked against the contract. Concerning the protocol state machines, concurrency is currently not supported, and operation call sequences which do not fit to the protocol are rejected. The definition of protocol state machines is optional.

Various validation and verification use cases for the USE tool are discussed in [9]. A comparison between the USE verification method for behavioral aspects and another approach is discussed in [11]. The so-called 'filmstripping' technique within USE for mapping behavioral descriptions into structural problems is proposed in [8].

5 UML Metamodel for Interactions

The interactions part of the UML metamodel[2] [19, p. 473ff.] was developed to visualize concrete traces of event occurrences and in addition to allow the definition of all possible traces of an interaction. The former can be used in early design stages to be able to communicate with designers and to some extent with stakeholders. A concrete trace does not show alternatives or loop constructs, because it describes a single message trace (or command trace) in the system. Elements like alternatives or loops can be used in later design phases to express all possible traces (cf. [19, p. 473]). Interactions can be visualized by different diagrams. Two of the more common ones are sequence diagrams and communication diagrams. Both diagrams focus on slightly different aspects of interactions. While sequence diagrams highlight the time line of an interaction, communication diagrams focus on the different elements participating in an interaction and their relationship.

Figure 6 shows an excerpt of the UML metamodel required to briefly discuss the representation of event occurrences inside interaction diagrams. A more detailed presentation can, for example, be found in [16]. On the right side of this figure, meta classes from the structural modeling part of the UML are shown. These are needed to completely model message occurrences. On the left side, the relevant parts of the interaction meta classes are shown. Consider the occurrence of the message `enter(hh)` shown in the following sequence diagram in Fig. 3 and in the (following) communication diagram in Fig. 8. This part of both diagrams can be expressed as an object diagram of the metamodel, as it is done in Fig. 7. Again, on the right side the structural part is shown, e. g., the two classes which participate in the message occurrence: `Truck` as the class of the receiving in-

[2] UML metamodel novices might skip this section on first reading and continue with the next section. UML metamodel followers are invited to dive deep.

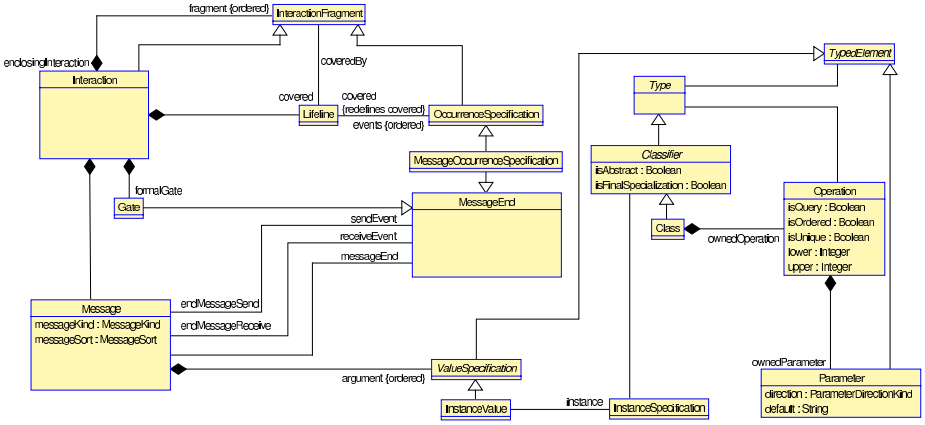


Fig. 6. Relevant parts of the UML interactions metamodel.

stance[3] and `Point` which is used as the type of the parameter of the operation `enter`. Further, both instances used in the interaction diagrams (`freds_scania` and `hh`) are placed there, too. On the left side, the example scenario is given as an instance of `Interaction`. Since we consider the single message occurrence `enter(hh)`, the object diagram contains few interaction related instances. First, the `Gate gSend` acts as the source of the message occurrence. It is linked to the interaction as a formal gate to signal that the source of the event is outside of this interaction. The receiving end of the message is represented by the instance `recEnter` of type `MessageOccurenceSpecification`. This instance is linked to the `Lifeline` named `freds_scania:Truck`. The payload of the message `mEnter` is given by the `InstanceValue` argument linked to the instance `hh` of the class `Point`.

6 Sequence Diagrams

As USE allows the developer to employ UML protocol machines to restrict the model behavior and to document test scenarios with sequence diagrams, it is desirable to show the protocol machine state of objects on sequence diagram lifelines, when the developer thinks this may be useful. Thus we have implemented this option for lifelines.

In Fig. 3, a fraction of the test scenario from Fig. 4 is displayed. We have manually selected the lifeline of only two `Point` objects and one `Truck` object and have activated the display of states from protocol state machines. For example, one can directly trace the development of the `Truck` object and the state changing

[3] In the current version of the UML metamodel, a lifeline can only represent connectable elements like properties or parameters. Since our tool allows a lifeline to represent a concrete instance, this fact cannot be expressed using the current UML metamodel. This is an open issue reported to the OMG [5].

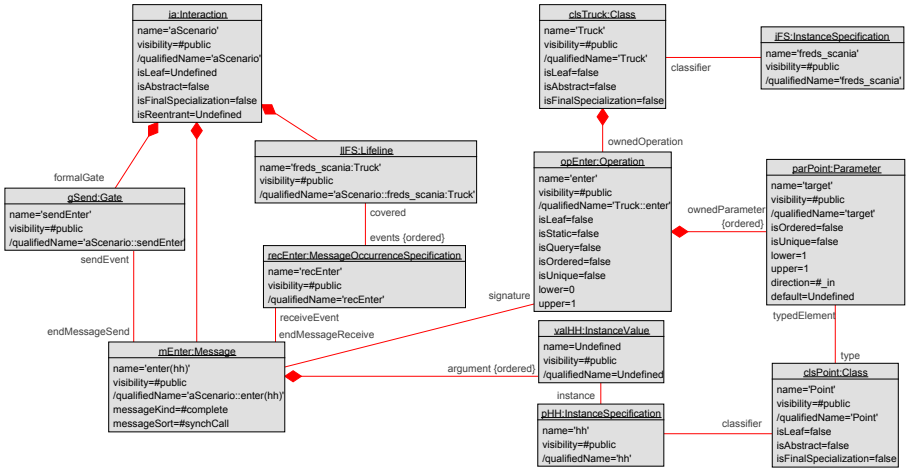


Fig. 7. Send message event as an instance of the UML metamodel.

through operation calls with `init(..)`, `enter(..)`, `move(..)`, `pay(..)`, `bye()`: from `born` to `noDebt` to `debt` and then again to `noDebt`. In the case that more money has been paid than is needed for paying the journey, the operation `bye` returns the overpayment.

UML sequence diagrams also allow the developer to use combined fragments, which define a combination of interaction fragments. A combined fragment consists of an interaction operator, an appropriate interaction operands and, if required, so-called guards (Boolean expressions).

Altogether, the UML supports 12 interaction operators. Some of these operators could be introduced in USE by representing SOIL operations as sequence diagrams. The *alternatives* and *option* operators, for example, could be realized via SOIL's conditional execution support (*if-then-else*). And the *loop* operator could be implemented via the SOIL iteration statement (*for-in-do-end*).

Sequence diagrams also support *interaction use* elements, which allow developers to call other interactions to simplify or reuse shared interactions. This could be represented in SOIL with corresponding operation calls, thus covering the *reference* interaction operator.

7 Communication Diagrams

Figure 8 shows the communication diagram representing the messages from the test scenario in Fig. 4 and additionally all messages that are executed within the operation calls by the SOIL implementation. As usual in communication diagrams, the ordering of messages is determined by message numbers, and submessages (i.e., messages that are triggered by one message) are displayed by a structured message number with a dot as separator. For example, message 18 has

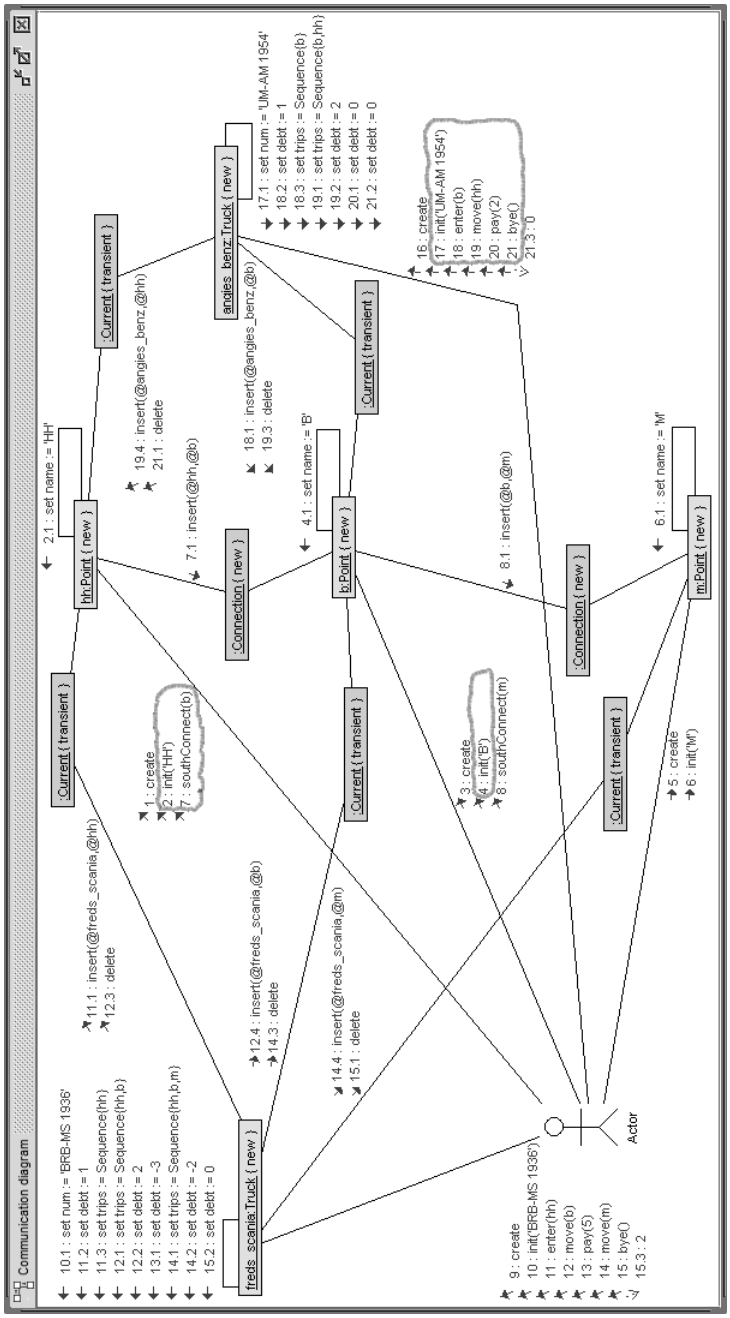


Fig. 8. Communication diagram with details shown (framed messages also in Fig 3).

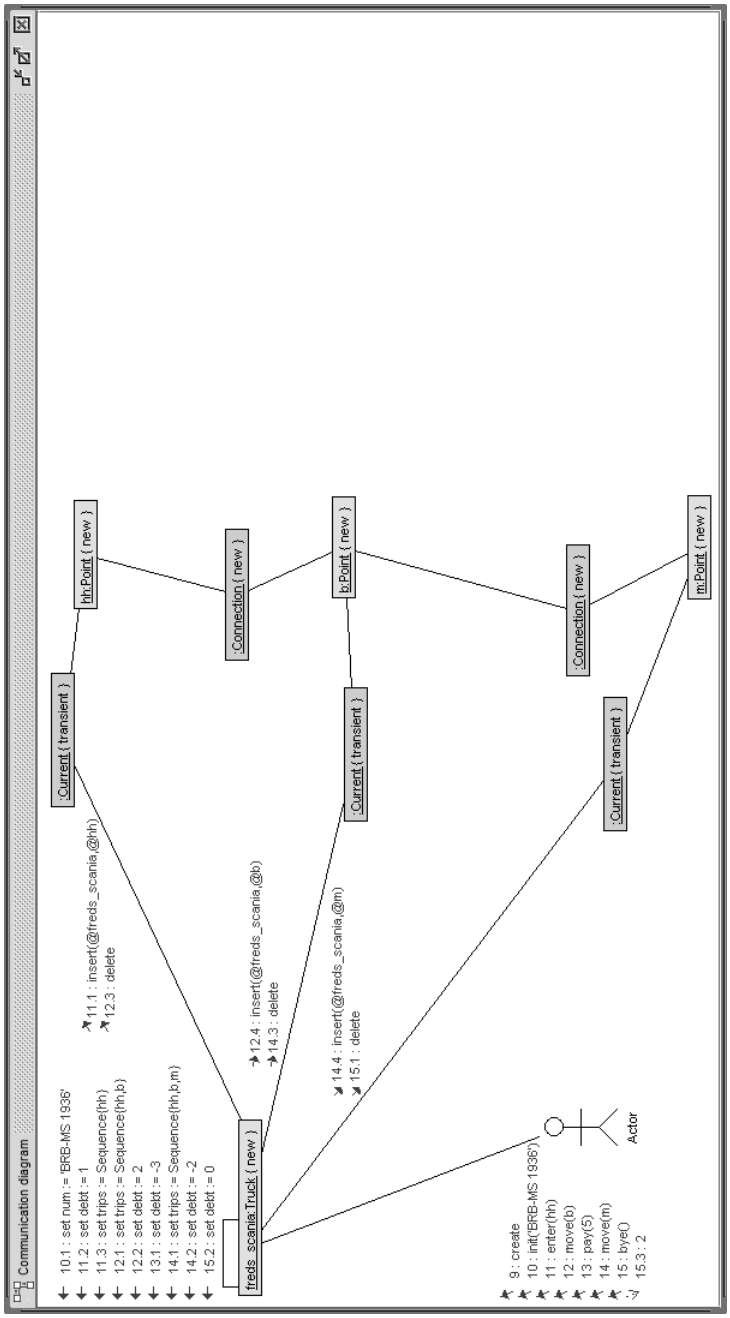


Fig. 9. Communication diagram displaying only messages 9-15.

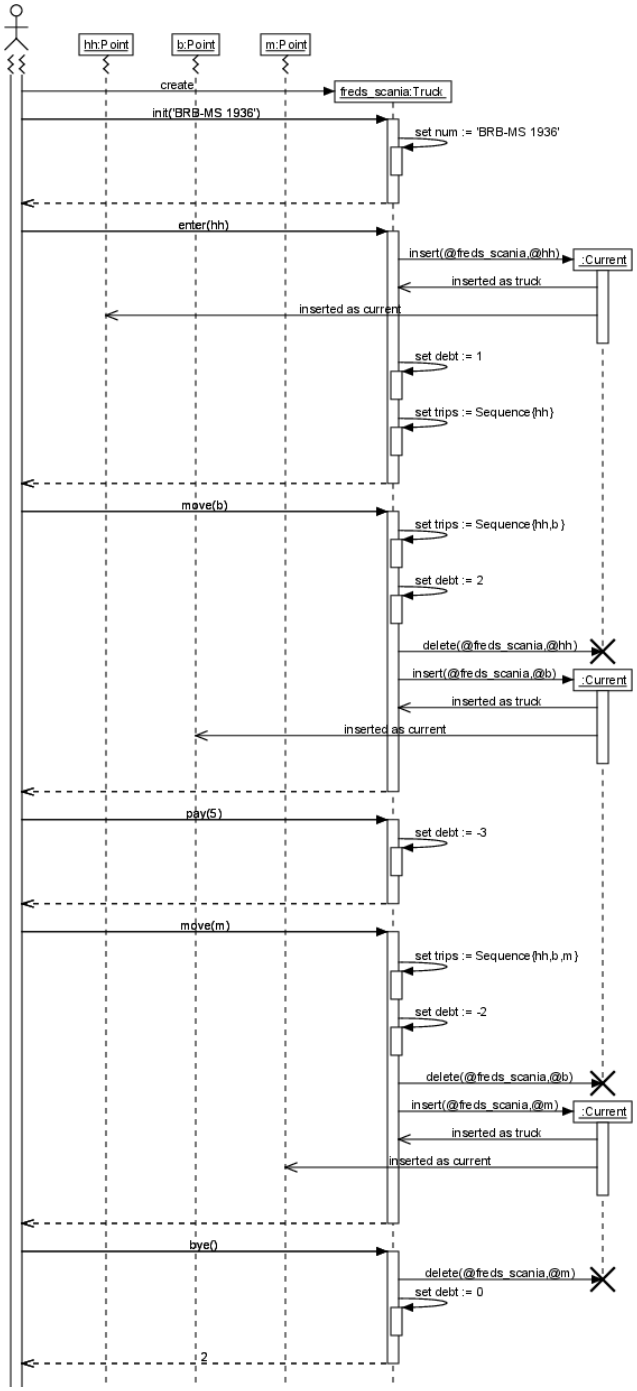


Fig. 10. Sequence diagram displaying only messages 9-15.

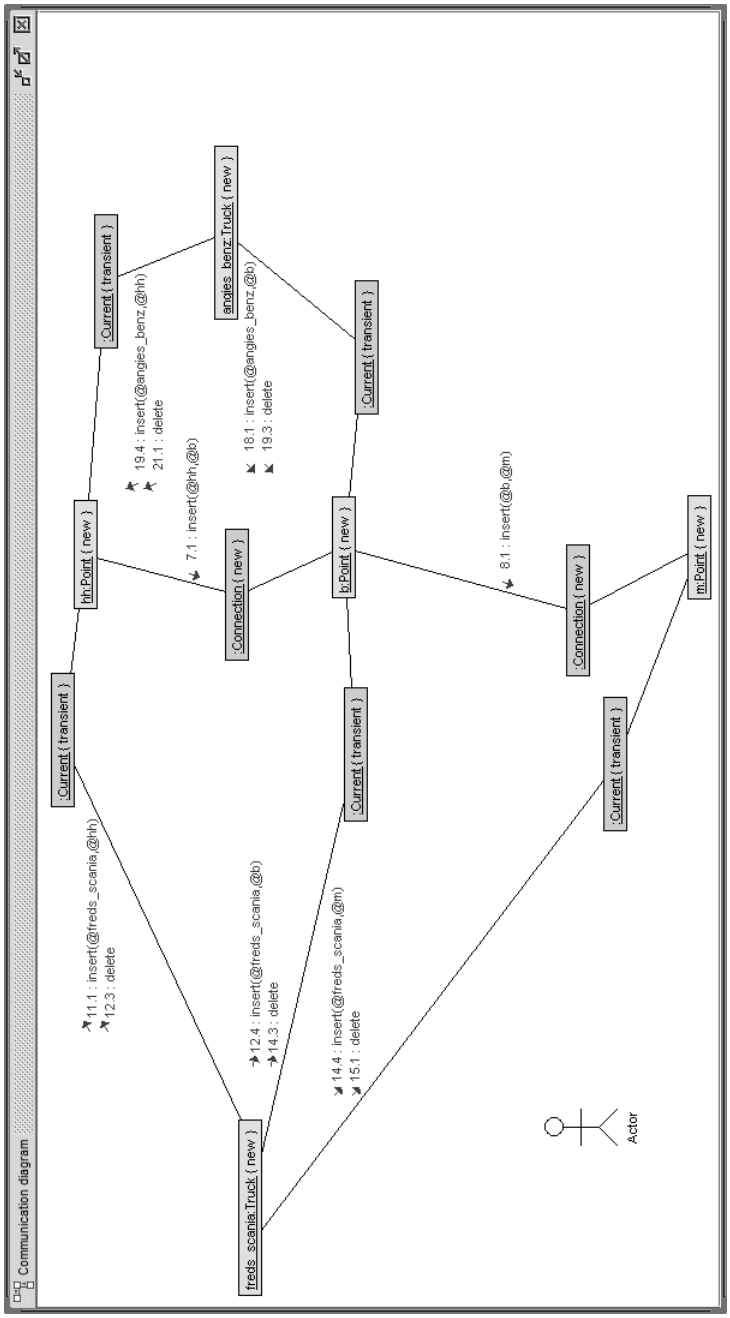


Fig. 11. Communication diagram displaying only link insertion and deletion.

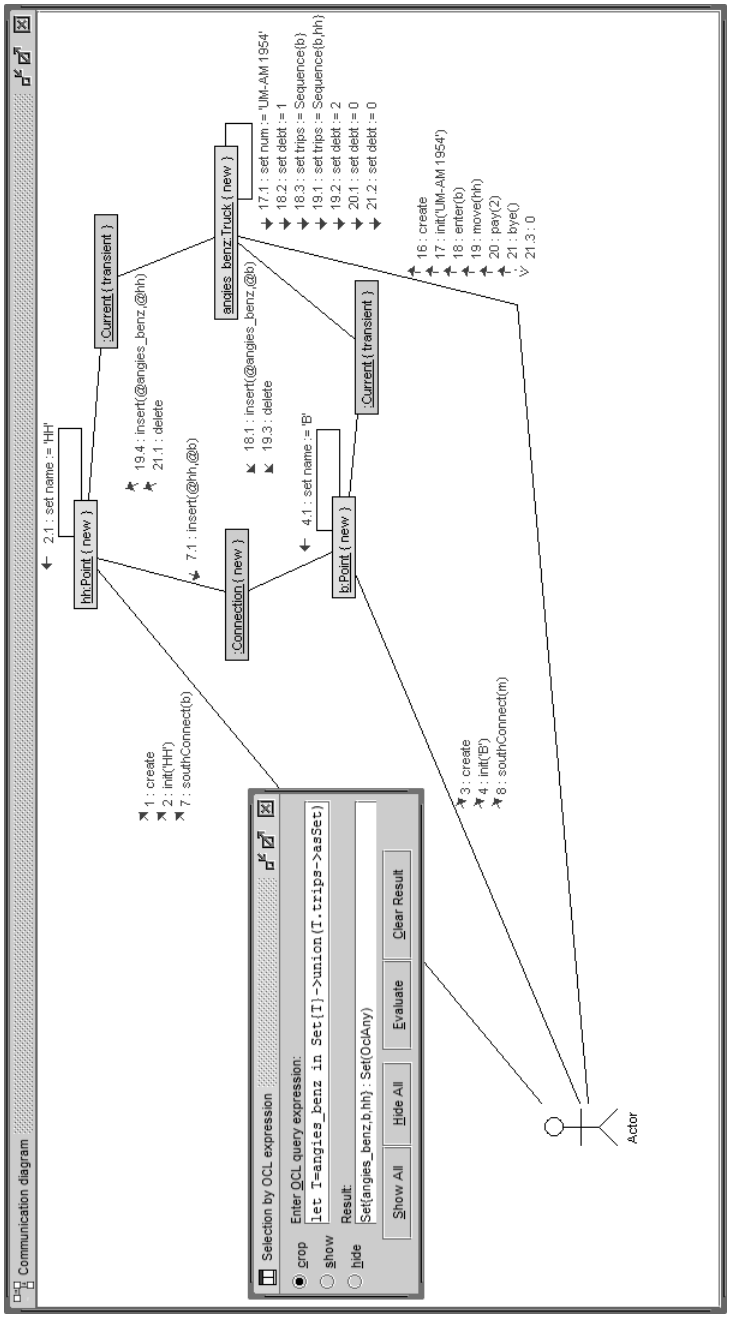


Fig. 12. Communication diagram with OCL selection for truck object by identity.

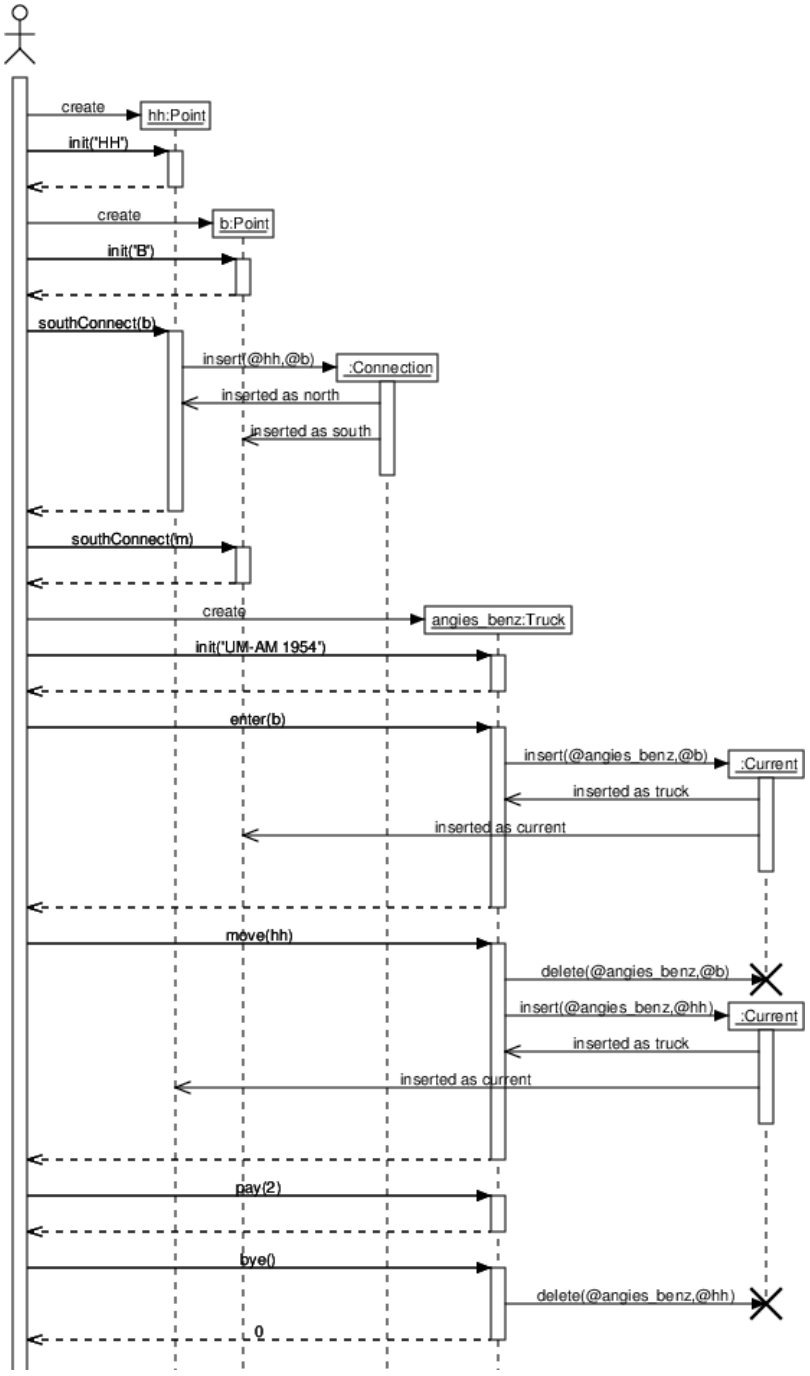


Fig. 13. Sequence diagram corresponding to communication diagram in Fig. 12.

the sub-messages 18.1, 18.2, and 18.3, i.e., the `enter(b)` call on the `Truck` object `angies_benz` is implemented by a link insertion (18.1) in association `Current`, an assignment (18.2) for attribute `debt` and an assignment (18.3) for attribute `trips`. As usual in communication diagrams, the specifications `new`, `transient`, resp. `destroyed` refer to objects that are newly introduced, newly introduced and deleted, resp. deleted during the interaction.

The relationship to the sequence diagram in Fig. 3 has been indicated manually by messages that are lying inside free drawn frames. These eight framed messages correspond to the eight messages in the sequence diagram.

For a smart representation of a communication diagram in an interactive GUI, the main objective is to provide a good overview and comprehensibility of the diagram. Bigger communication diagrams with multiple operation calls and messages become quickly difficult to follow (see Fig. 8). To improve this situation, some straightforward ideas have proven to be helpful:

1. Limiting the view of the diagram to a range of messages (see Fig. 9).
2. Cropping of different message types to only display those messages that are relevant to understand the shown process (see Fig. 11).
3. Cropping of objects and links to only display those relevant in the shown process (see Fig. 12).
4. Combinations of the above.

The communication diagram in Fig. 8 shows the complete sequence of messages (1–21), which can be roughly split into the *initialization of a road network* and two *navigations of trucks*. Figure 9 focuses on the navigation of the first truck only (messages 9–15) and thereby this sequence is easier to understand.

A similar effect occurs when focusing on a subset of message types. Figure 11 only shows link insertion and deletion messages in the communication diagram and thereby increases the focus on the development of the links. A similar feature is available for sequence diagrams, allowing to show or hide the message types `create`, `destroy`, `insert`, `delete` and `set`.

Lastly, single objects and links that are not relevant to understand the current process can be removed from the view of the diagram in favor of a better accessibility, e.g., in Fig. 12 only one truck, the two points that it visits and the links in between these objects are displayed. The other parts of the route as well as the second truck are hidden.

Thus, to help with the selection of large quantities of objects communication diagrams, the *selection by OCL expression* feature of the USE tool has been taken over from the object diagram (see Fig. 12). With this feature, certain objects can be shown, hidden or cropped.

8 Selection Mechanisms in Communication and Sequence Diagrams

To further illustrate and compare the selection mechanisms in sequence and communication diagrams, the following three examples demonstrate selecting views on the complete interaction from Fig. 8 where one particular aspect is emphasized in each example. Where appropriate, the corresponding sequence diagram is also displayed with the same filters applied.

1. **Interval selection:** Figure 9 restricts the messages according to a message number interval: only the messages 9 to 15 including their sub-messages are stated. This part of the interaction handles the first `Truck` object and shows its initialization and movements. Figure 10 shows the corresponding sequence diagram with the same selection applied.
2. **Message kind selection:** Figure 11 presents a view on the complete interaction along a different dimension than message numbers. Only messages concerning a particular message kind are displayed, in this diagram the insertion and deletion of links. As in UML different message kinds are available, such a restriction can be useful. In USE we currently support the following message kinds: object creation, object destruction, link creation, link destruction, attribute assignment, and operation call.
3. **OCL selection:** Figure 12 makes a selection in the communication diagram with the help of an OCL expression. In this case the OCL expression picks a `Truck` object together with the `Point` objects that are visited. The result is typed as `Set(OclAny)` because objects of different classes show up. All messages between the selected objects are shown. This object and message selection cannot be achieved with a message number interval or a message kind specification. Figure 13 shows the corresponding sequence diagram with the same selection applied, however `set` statements are hidden.

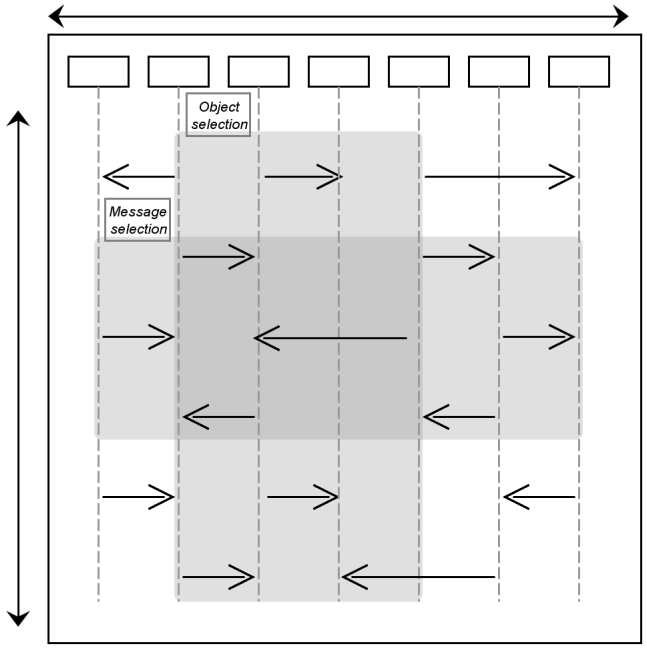
The selection mechanisms shown in the communication diagrams in Figs. 8 and 12, are currently implemented (modulo some required improvements in the user interface). USE also supports the selection mechanisms shown in Figs. 9 and 11.

9 Systematic Selection Mechanisms for Views in UML Interactions and Further Use of OCL

Currently, the selection mechanisms for UML sequence and communication diagrams in our tool USE are different. This is due to the fact that the design and implementation has been done at different times with different people involved. Our plan is to unify the selection mechanisms and offer a unified view mechanism for both interaction diagrams. We currently identify the following options. An overview in form of a generic interaction together with the object and message dimensions and the resulting presentation options is presented in Fig. 14.

Model behavior determined by
- Class diagram with class invariants and operation pre- and postconditions
- Statecharts with state invariants and transition pre- and postconditions

Object selection through
- Manual selection or selection by enumeration
- Class selection (objects of particular classes)
- Operation selection (objects sending/receiving calls of particular operations)
- OCL expression



Message selection through
- Manual selection or selection by enumeration
- Class selection (messages in particular classes)
- Operation selection (messages of particular operations)
- Message interval
- Message kind
- Message depth

Fig. 14. Overview on interactions with object and message dimension.

Selection focusing on objects: Objects could be selected through the following possibilities:

1. Interactive show, hide or crop for objects individually or by class.
2. Interactive multiple selection by shift key and mouse click.
3. Objects satisfying resp. violating an OCL invariant during interaction.
4. Objects satisfying resp. violating an ad-hoc OCL formula during interaction.

Selection focusing on messages: Messages could be selected through the following possibilities:

1. Interactive show, hide or crop for messages.
2. Selection through an OCL object query identifying the sending object.
3. Selection through a satisfied resp. violated OCL pre- or postcondition.
4. Selection through a satisfied resp. violated ad-hoc OCL formula at pre- or postcondition time during an operation call.
5. Selection by message kind: object creation, object destruction, link insertion, link deletion, attribute assignment, operation call.
6. Selection by message number depth.
7. Determination of a message interval defined by
   (a) interactively fixed start message and end message.
   (b) start OCL formula and end OCL formula.
   (c) a statechart start state and a statechart end state for a fixed object.

The OCL expressions that we employ in communication diagrams are currently working on the last system state. However, the communication diagram contains information that is not selectable using plain OCL in this way, i.e., removed objects and links in general. For example the OCL expression `allInstances()` to select all instances of a class will currently not select transient or destroyed objects, yet they are still displayed in the communication diagram.

Consequently, to get full access to the elements in the communication diagram, the syntax and accordingly the evaluation of OCL has to be extended. First, it is desired to access the system's pre- and post states of each message to get access to all time steps of the communication diagram. In addition, access to the elements of a range of messages or the global sequence of messages is helpful for the selection. Temporal extensions for OCL often include functionality to formulate expression about the past (see e.g., [25]) and can be considered to be integrated.

The temporal extension of OCL would not only improve the selection of elements in the GUI. The access to the new properties increases the possibilities of validation tasks formulated on the communication diagram.

10 Related Work

Behavior modeling with UML interactions has relationships to other important approaches. A definition of the UML interaction semantics in terms of the System Model can be found in [3]. In [13], a comparison between software model

verification approaches using OCL and UML interaction diagrams among others is performed. The work in [17] focuses on the interaction problem in the context of aspect-oriented programming. It explains how Aspect-UML can be translated into Alloy and shows how to verify aspect interactions with Alloy's model analyzer. In [18], the synthesis of test cases from UML interaction diagrams by a systematic interpretation of flow of controls is discussed. Improvements to the UML interaction metamodel concerning message arguments and loops are proposed and demonstrated in [24]. The approach in [15] is strongly related to the USE approach because of the emphasis on protocol modeling. That work is however closer to programming through the use of Java, whereas we are closer to modeling because of using OCL. The proposals in [4, 14] discuss test case generation from interaction diagrams. Our approach is the only one that employs OCL for selecting relevant parts in the interactions under consideration. The current work differs from our previous contributions (like [7, 10]) in that we did not consider sequence diagrams with statechart states on lifelines or communication diagrams at all.

11 Conclusion

This contribution has discussed how to handle UML interaction diagrams in a model validation tool and has pointed to the link between protocol machine and interaction diagrams. We have set up desirable selection mechanisms for both kinds of UML interaction diagrams, namely sequence and communication diagrams.

Future work has to complete our current implementation with the missing features in both interaction diagrams. In particular, message kind selection and message interval selection seem to offer useful analysis options. We have discussed how to extend the options for interaction analysis with temporal OCL query features. Larger examples and case studies need to validate the already existing and planned features for better support of interaction diagrams that advance behavioral modeling.

References

1. F. Büttner, U. Bartels, L. Hamann, O. Hofrichter, M. Kuhlmann, M. Gogolla, L. Rabe, F. Steimke, Y. Rabenstein, and A. Stosiek. Model-Driven Standardization of Public Authority Data Interchange. *Science of Computer Programming*, 89:162–175, 2014.
2. F. Büttner and M. Gogolla. Modular Embedding of the Object Constraint Language into a Programming Language. In A. Simao and C. Morgan, editors, *Proc. 14th Brazilian Symposium on Formal Methods (SBMF'2011)*, pages 124–139. Springer, Berlin, LNCS 7021, 2011.
3. D. Calegari, M. V. Cengarle, and N. Szasz. UML 2.0 Interactions with OCL/RT Constraints. In *FDL*, pages 167–172. IEEE, 2008.

4. H. Y. Chen, C. Li, and T. H. Tse. Transformation of UML Interaction Diagrams into Contract Specifications for Object-oriented Testing. In IEEE [12], pages 1298–1303.
5. M. M. J. Chonoles. Issue 15123: Sequence Diagram and Communication Diagrams should Support Instances as Lifelines (uml2-rtf), Mar. 2010. `http://www.omg.org/issues/uml2-rtf.html#Issue15123`.
6. G. Georg and R. France. An Activity Theory Language: USE Implementation. Colorado State University, Computer Science, Technical Report CS-13-101, 2013.
7. M. Gogolla, F. Büttner, and M. Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69:27–34, 2007.
8. M. Gogolla, L. Hamann, F. Hilken, M. Kuhlmann, and R. B. France. From Application Models to Filmstrip Models: An Approach to Automatic Validation of Model Dynamics. In H. Fill, D. Karagiannis, and U. Reimer, editors, *Proc. Modellierung (MODELLIERUNG'2014)*, pages 273–288. GI, LNI 225, 2014.
9. M. Gogolla, M. Kuhlmann, and L. Hamann. Consistency, Independence and Consequences in UML and OCL Models. In C. Dubois, editor, *Proc. 3rd Int. Conf. Test and Proof (TAP'2009)*, pages 90–104. Springer, Berlin, LNCS 5668, 2009.
10. L. Hamann, O. Hofrichter, and M. Gogolla. Towards Integrated Structure and Behavior Modeling with OCL. In R. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *Proc. 15th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'2012)*, pages 235–251. Springer, Berlin, LNCS 7590, 2012.
11. F. Hilken, P. Niemann, M. Gogolla, and R. Wille. Filmstripping and Unrolling: A Comparison of Verification Approaches for UML and OCL Behavioral Models. In M. Seidl and N. Tillmann, editors, *Proc. 8th Int. Conf. Tests and Proofs (TAP 2014)*, pages 99–116. Springer, LNCS 8570, 2014.
12. IEEE, editor. *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Montréal, Canada, 7-10 October 2007*. IEEE, 2007.
13. A. Knapp and J. Wuttke. Model Checking of UML 2.0 Interactions. In T. Kühne, editor, *MoDELS Workshops*, volume 4364 of *Lecture Notes in Computer Science*, pages 42–51. Springer, 2006.
14. P. D. L. Machado, J. C. A. de Figueiredo, E. F. A. Lima, A. E. V. Barbosa, and H. S. Lima. Component-based Integration Testing from UML Interaction Diagrams. In IEEE [12], pages 2679–2686.
15. A. T. McNeile and N. Simons. Protocol Modelling: A Modelling Approach that Supports Reusable Behavioural Abstractions. *Software and System Modeling*, 5(1):91–107, 2006.
16. Z. Micskei and H. Waeselynck. The Many Meanings of UML2 Sequence Diagrams: A Survey. *Software & Systems Modeling*, 10(4):489–514, 2011.
17. F. Mostefaoui and J. Vachon. Design-Level Detection of Interactions in Aspect-UML Models Using Alloy. *Journal of Object Technology*, 6(7):137–165, 2007.
18. A. Nayak and D. Samanta. Model-based Test Cases Synthesis using UML Interaction Diagrams. *ACM SIGSOFT Software Engineering Notes*, 34(2):1–10, 2009.
19. OMG, editor. *UML Superstructure 2.4.1*. Object Management Group (OMG), Aug. 2011.
20. OMG, editor. *Object Constraint Language, Version 2.3.1*. OMG, 2012. OMG Document, `www.omg.org`.
21. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language 2.0 Reference Manual*. Addison-Wesley, Reading, 2003.
22. B. Selic. The Theory and Practice of Modeling Language Design. Tutorial at MODELS 2012, `http://models2012.info/`, 2012.

23. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 2003. 2nd Edition.
24. M.-F. Wendland, M. Schneider, and Ø. Haugen. Evolution of the UML Interactions Metamodel. In A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. J. Clarke, editors, *MoDELS*, volume 8107 of *Lecture Notes in Computer Science*, pages 405–421. Springer, 2013.
25. P. Ziemann and M. Gogolla. OCL Extended with Temporal Logic. In M. Broy and A. Zamulin, editors, *5th Int. Conf. Perspectives of System Informatics (PSI'2003)*, pages 351–357. Springer, Berlin, LNCS 2890, 2003.

Appendix: Complete USE model for Toll Collect

```
-------------------------------------------------- model TollCollect
model TollCollect
------------------------------------------------------- class Truck
class Truck
attributes
  num:String            init: ''
  trips:Sequence(Point) init: Sequence{}
  debt:Integer          init: 0
operations
  init(aNum:String)
    begin self.num:=aNum end
  enter(entry:Point)
    begin insert (self,entry) into Current; self.debt:=1;
    self.trips:=self.trips->including(self.current) end
  move(target:Point)
    begin self.trips:=self.trips->including(target);
    self.debt:=self.debt+1; delete (self,self.current) from Current;
    insert (self,target) into Current end
  pay(amount:Integer)
    begin self.debt:=self.debt-amount end
  bye():Integer
    begin delete (self,self.current) from Current;
    result:=self.debt.abs(); self.debt:=0 end
  ------------------------------------------------------------------
  numIsKey():Boolean=
    Truck.allInstances->forAll(self,self2|
      self<>self2 implies self.num<>self2.num)
  ------------------------------------------------------------------

statemachines
  psm TruckLife
  states
    prenatal:initial
    born   [num='']
    noDebt [num<>'' and current->isEmpty]
    debt   [num<>'' and current->notEmpty]
```

```
  transitions
    prenatal -> born   { create }
    born     -> noDebt { init() }
    noDebt   -> debt   { enter() }
    debt     -> debt   { move() }
    debt     -> debt   { pay() }
    debt     -> noDebt { bye() }
  end
end

------------------------------------------------------- class Point
class Point
attributes
  name:String init: ''
  isJunction:Boolean derived: north->union(south)->size()>=2
operations
  init(aName:String)
    begin self.name:=aName end
  northConnect(aNorth:Point)
    begin insert (aNorth,self) into Connection end
  southConnect(aSouth:Point)
    begin insert (self,aSouth) into Connection end
  ---------------------------------------------------------------
  northPlus():Set(Point)=north->closure(p|p.north)
  southPlus():Set(Point)=south->closure(p|p.south)
  ---------------------------------------------------------------
  nameIsKey():Boolean=
    Point.allInstances->forAll(self,self2|
      self<>self2 implies self.name<>self2.name)
  noCycles():Boolean=
    Point.allInstances->forAll(self|
      not(self.northPlus()->includes(self)))
  ---------------------------------------------------------------

statemachines
  psm PointLife
  states
    prenatal:initial
    born    [name='']
    growing [name<>'']
  transitions
    prenatal -> born    { create }
    born     -> growing { init() }
    growing  -> growing { northConnect() }
    growing  -> growing { southConnect() }
  end
end
```

```
-------------------------------------------- association Current
association Current between
  Truck[0..*] role truck
  Point[0..1] role current
end
-------------------------------------------- association Connection
association Connection between
  Point[0..*] role north
  Point[0..*] role south
end

------------------------------------------------------ constraints
constraints
------------------------------------------------------ invariants
context Truck inv numIsKeyInv:
  numIsKey()
context Point inv nameIsKeyInv:
  nameIsKey()
context Point inv noCyclesInv:
  noCycles()

------------------------------------------------------ Point::init
context Point::init(aName:String)
pre freshPoint:
  self.name='' and self.north->isEmpty and self.south->isEmpty
pre aNameOk:
  aName<>'' and aName<>null
post nameAssigned:
  aName=self.name
post allPointInvs:
  nameIsKey() and noCycles()

------------------------------------------------ Point::northConnect
context Point::northConnect(aNorth:Point)
pre aNorthDefined:
  aNorth.isDefined
pre freshConnection:
  self.north->excludes(aNorth) and self.south->excludes(aNorth)
pre notSelfLink:
  self<>aNorth
pre noCycleIntroduced:
  aNorth.northPlus()->excludes(self)
post connectionAssigned:
  self.north->includes(aNorth)
post allPointInvs:
  nameIsKey() and noCycles()
```

```
------------------------------------------------------- Truck::init
context Point::southConnect(aSouth:Point)
pre aSouthDefined:
  aSouth.isDefined
pre freshConnection:
  self.south->excludes(aSouth) and self.south->excludes(aSouth)
pre notSelfLink:
  self<>aSouth
pre noCycleIntroduced:
  aSouth.southPlus()->excludes(self)
post connectionAssigned:
  self.south->includes(aSouth)
post allPointInvs:
  nameIsKey() and noCycles()

------------------------------------------------------- Truck::init
context Truck::init(aNum:String)
pre freshTruck:
  self.num='' and self.trips=Sequence{} and self.debt=0 and
  self.current->isEmpty
pre aNumOk:
  aNum<>'' and aNum<>null
post numAssigned:
  aNum=self.num
post allTruckInvs:
  numIsKey()

------------------------------------------------------- Truck::enter
context Truck::enter(entry:Point)
pre noDebt:
  0=self.debt
pre currentEmpty:
  self.current->isEmpty
pre entryOk:
  entry<>null
post debtAssigned:
  1=self.debt
post currentAssigned:
  entry=self.current
post allTruckInvs:
  numIsKey()
```

```
------------------------------------------------------- Truck::move
context Truck::move(target:Point)
pre currentExists:
  self.current->notEmpty
pre targetReachable:
  self.current.north->union(self.current.south)->includes(target)
post debtIncreased:
  self.debt@pre+1=self.debt
post tripsUpdated:
  self.trips@pre->including(target)=self.trips
post currentAssigned:
  target=self.current
post allTruckInvs:
  numIsKey()

-------------------------------------------------------- Truck::pay
context Truck::pay(amount:Integer)
pre amountPositive:
  amount>0
pre currentExists:
  self.current->notEmpty
post debtReduced:
  (self.debt@pre-amount)=(self.debt)
post allTruckInvs:
  numIsKey()

-------------------------------------------------------- Truck::bye
context Truck::bye():Integer
pre currentExists:
  self.current->notEmpty
pre noDebt:
  self.debt<=0
post resultEqualsOverPayment:
  self.debt@pre.abs()=result
post zeroDebt:
  self.debt=0
post currentEmpty:
  self.current->isEmpty
post allTruckInvs:
  numIsKey()
-----------------------------------------------------------------------
```