Behavior Modeling with Interaction Diagrams in a UML and OCL Tool

Martin Gogolla, Lars Hamann, Frank Hilken, Matthias Sedlmeier, Quang Dung Nguyen
University of Bremen, Germany
{gogolla,lhamann,fhilken,ms,quang}@tzi.de

ABSTRACT

This contribution 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 a systematic set of features which should be available in both kinds of interaction diagrams. We emphasize the role that OCL can play for such a feature set.

Categories and Subject Descriptors

D.2.2 [**Software engineering**]: Design Tools and Techniques—*Object-oriented design methods*; D.2.4 [**Software engineering**]: Software/Program Verification—*Validation*; F.3.1 [**Logics and meanings of programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Design, Languages, Verification

Keywords

UML, OCL, Model behavior, Statechart diagram, Interaction diagram, Sequence Diagram, Communication Diagram, Model validation, Diagram view

1. INTRODUCTION

The Unified Modeling language (UML) has become a de-facto standard for the graphical design of IT systems. UML [14; 16] 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 [15; 17] within UML.

This contribution focuses on UML interaction diagrams in form of sequence and communication diagrams. We introduce new features for interactions in a UML tool and discuss how the two interaction diagrams could be handled in uniform way.

Since about 15 years our group is developing the UML and OCL tool USE (UML-based Specification Environment). USE [5; 6] 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.

The structure of the rest of this paper is as follows. Section 2 introduces a running example. Section 3 shows the UML metamodel for interactions and sets the context for the interaction diagram implementation within USE. Section 4 presents new features in sequence diagrams, and Sect. 5 discusses communication diagrams. Section 6 proposes a systematic set of features that could be available in both interaction diagrams. Section 7 compares our approach to related papers. The contribution is closed in Sect. 8 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 sys-

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
BM-FA '14, July 22 2014, York, United Kingdom
Copyright 2014 ACM 978-1-4503-2791-6/14/07 $15.00.
http://dx.doi.org/10.1145/2630768.2630774.

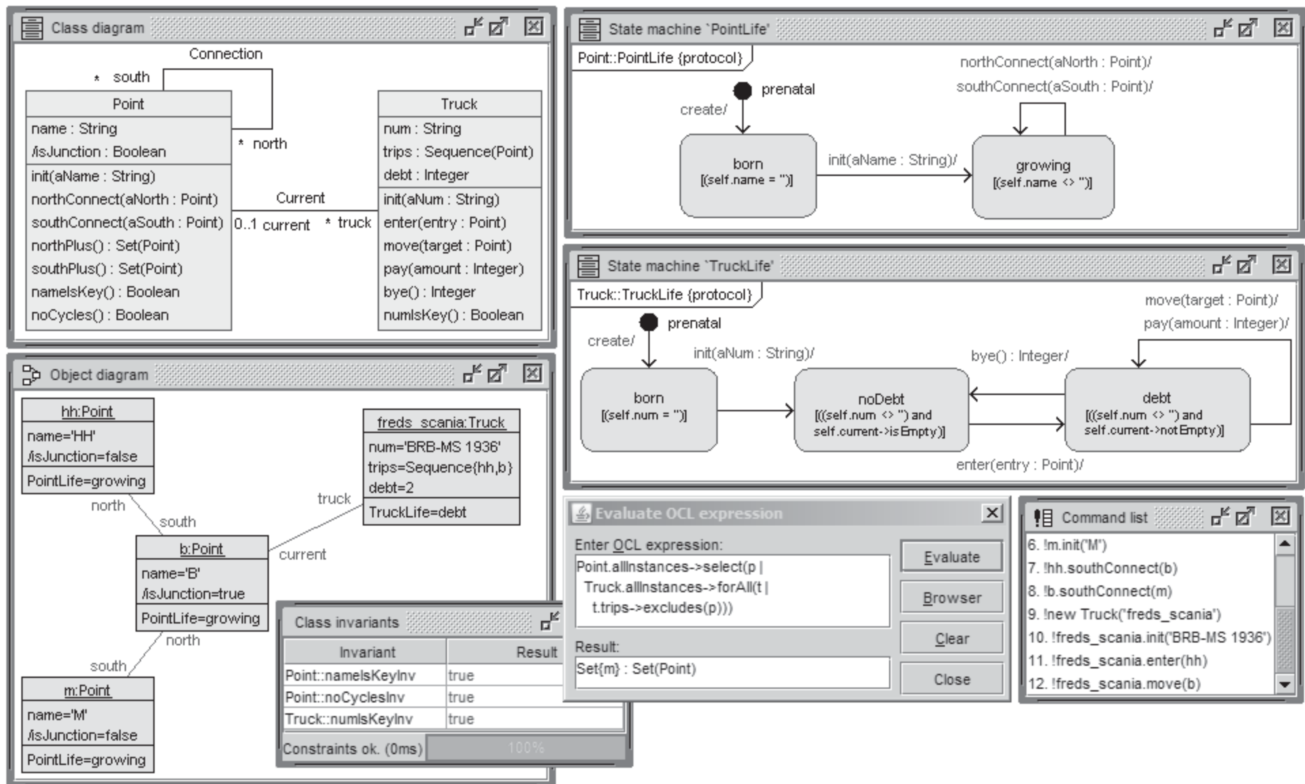

Figure 1: Example model Toll Collect.

```
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()
```

Figure 2: Example of operation implementation and pre- and postconditions.

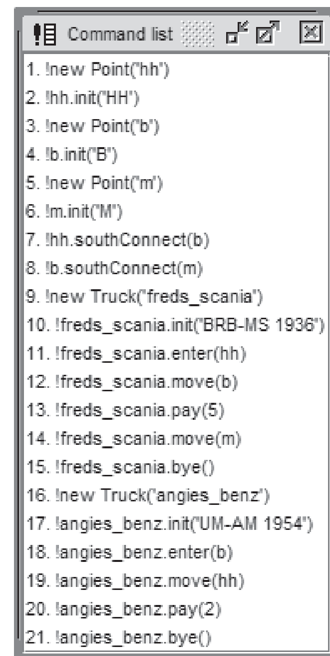

Figure 3: Command list for used interaction diagrams.

tem 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 implementation in SOIL [1] is shown in Fig. 2.

In Fig. 3, 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 slighty larger motorway example allowing to travel between western and eastern points as well is shown in Fig. 7. The complete USE model is given in the Appendix.

3. 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, and thus there is no need to include loops or conditionals, for example, in sequence diagrams.

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 [5]. 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 operations contracts, 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.

4. UML METAMODEL FOR INTERACTIONS

The interactions part of the UML metamodel[2] [14, 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 extend 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 (c. f. [14, 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 4 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 [11]. 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 forthcoming sequence diagram in Fig. 6 and in the (forthcoming) 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. 5. 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 instance[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 the source of the

[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.

[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 [4].

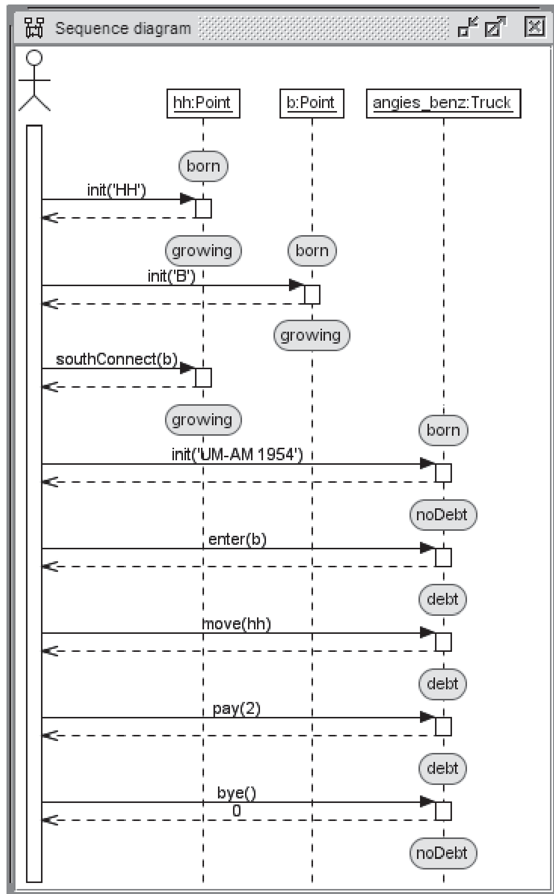

Figure 6: Sequence diagram with statechart states on lifeline (some details suppressed).

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`.

5. 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. 6, a fraction of the test scenario from Fig. 3 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 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.

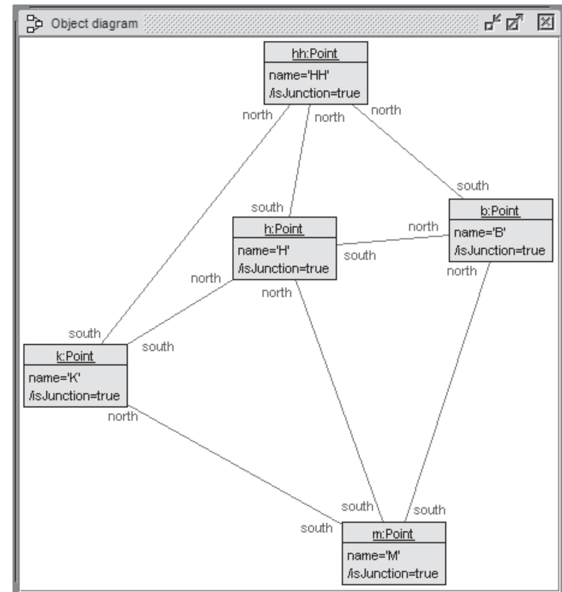

Figure 7: Example for motorway connections.

6. COMMUNICATION DIAGRAMS

Figure 8 shows the communication diagram representing the messages from the test scenario in Fig. 3 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 sub-messages (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 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 `dept` 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. 6 has been indicated manually by messages that are lying inside 'clouds'. These eight 'clouded' messages correspond to the eight messages in the sequence diagram.

The following five communication diagrams show selecting views on the complete interaction from Fig. 8 where one particular aspect is emphasized in each diagram.

1. Figure 9 restricts the messages according to a message number interval: only the messages 9 to 15 and its sub-messages are stated. This part of the interaction handles the first `Truck` object and shows its initialization and movements.

2. Figure 10 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

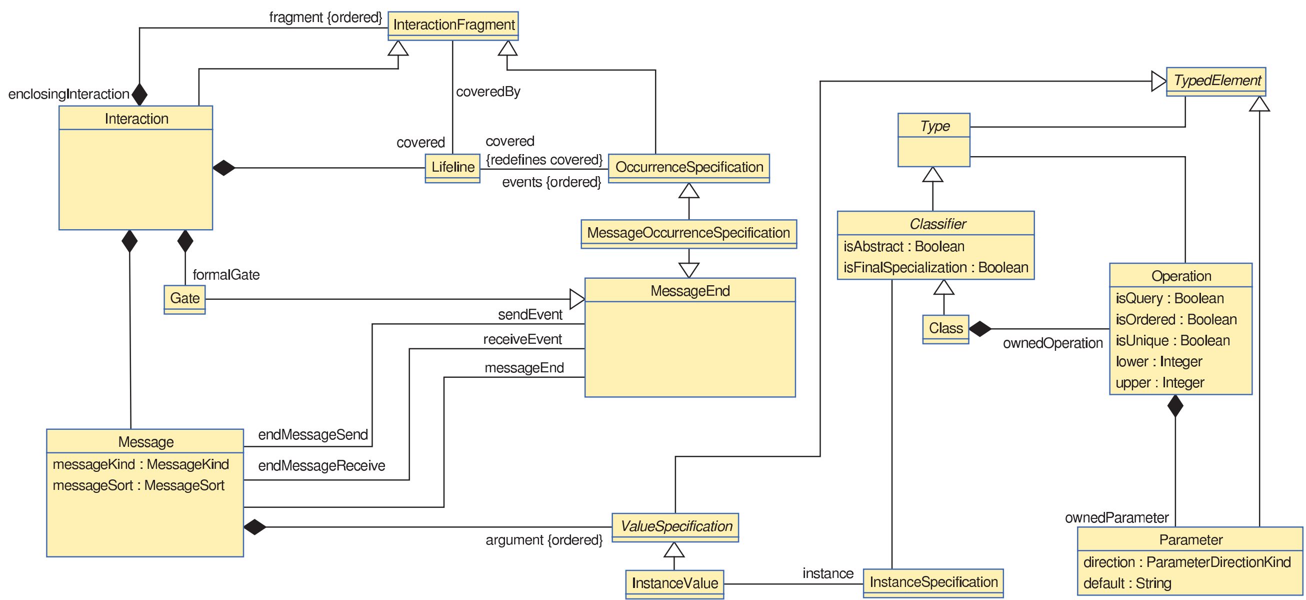

Figure 4: Relevant parts of the UML interactions metamodel.

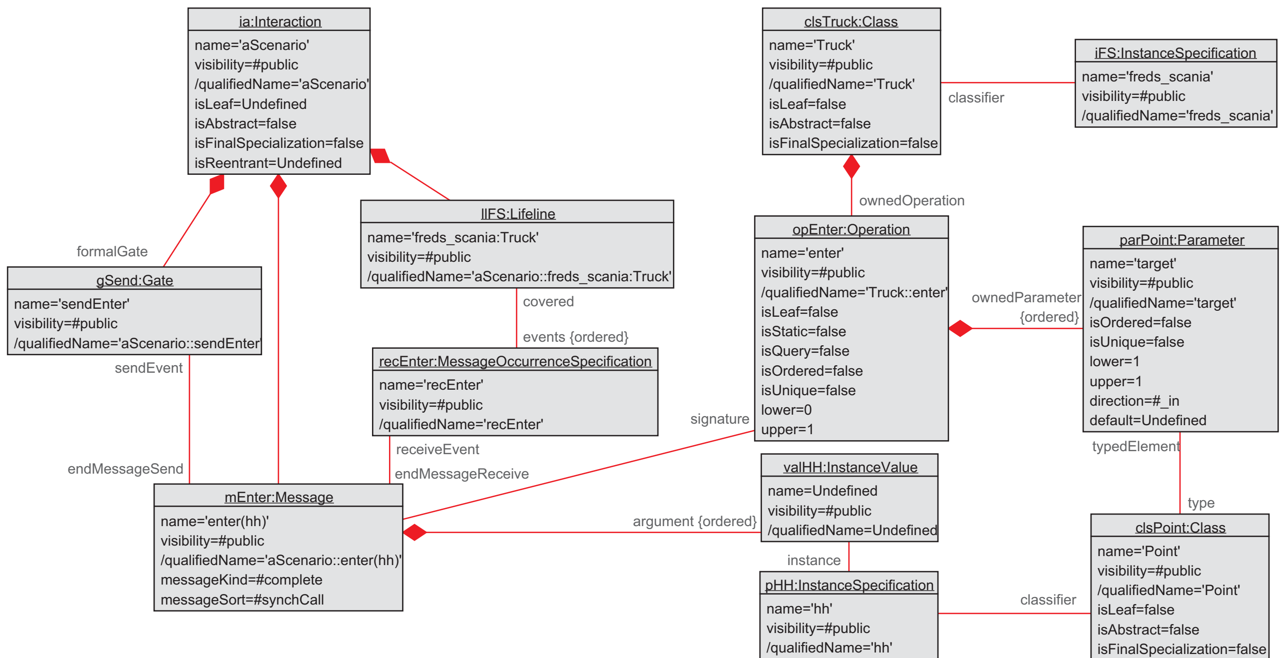

Figure 5: Send message event as an instance of the UML metamodel.

we currently support the following message kinds: object creation, object destruction, link creation, link destruction, attribute assignment, and operation call.

3. Figure 11 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.

4. Figure 12 gives a second example for an OCL selection. Here, a `Point` object is selected that serves as the end point for the trips of a first `Truck` and as a start point for the trips of a second `Truck`. Additionally, the trucks visiting the selected `Point` are shown.

5. Figure 13 is the third OCL selection in a communication diagram. Adjacent `Point` objects (adjacent with respect to the underlying motorway), on which at least two different trucks travel, are caught.

   The selection features shown in the communication diagrams in Figs. 8, 11, 12, and 13 are currently implemented (modulo some required improvements in the user interface). The features in Figs. 9 and 10 have to be implemented in future work.

7. SYSTEMATIC FEATURE SET FOR VIEWS IN UML INTERACTIONS AND FURTHER USE OF OCL

Currently, the selection features 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 features and offer a unified view mechanism for both interaction diagrams. We currently identify the following options.

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

1. Interactive hide or show for objects.
2. Objects satisfying resp. violating an OCL invariant during interaction.
3. 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 hide or show 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 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 semantics of OCL has to be extended. First, it is desired to access the system 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. [19]) 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. An example showing a validation task, already possible with the current form of the OCL selection is shown in Fig. 12. The query selects those navigation points, at which one truck finishes its trip and another truck begins its trip. Currently, this query utilizes the attribute `trips`, which saves the points a truck passes on the road network and thus make historical information available in the latest system state. With the extensions to OCL, this information is accessible without extra attributes in the communication diagram. To do so, the links that either are currently (in the latest system state) and those who were connected to the points previously could be analyzed together.

Another example is the selection of all points that are used multiple times. The corresponding communication diagram and the query is shown in Fig. 13. The functionality of the expression w.r.t. historical information is analogous to the previous example.

8. 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 [2]. In [8], a comparison between software model verification approaches using OCL and UML interaction diagrams among others is performed. The work in [12] focuses on the interaction problem in the context of aspect-oriented programming. It explains how Aspect-UML can be

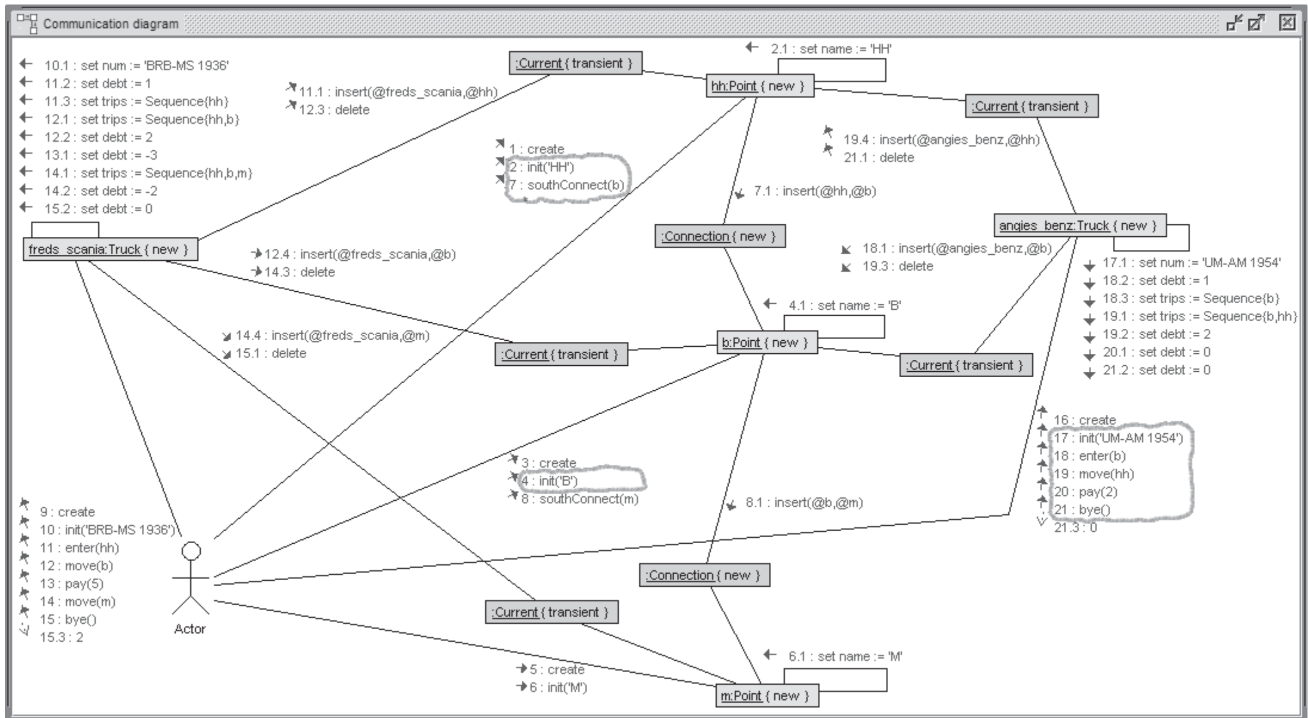

Figure 8: Communication diagram with all details displayed ('clouded' messages also in Fig 6).

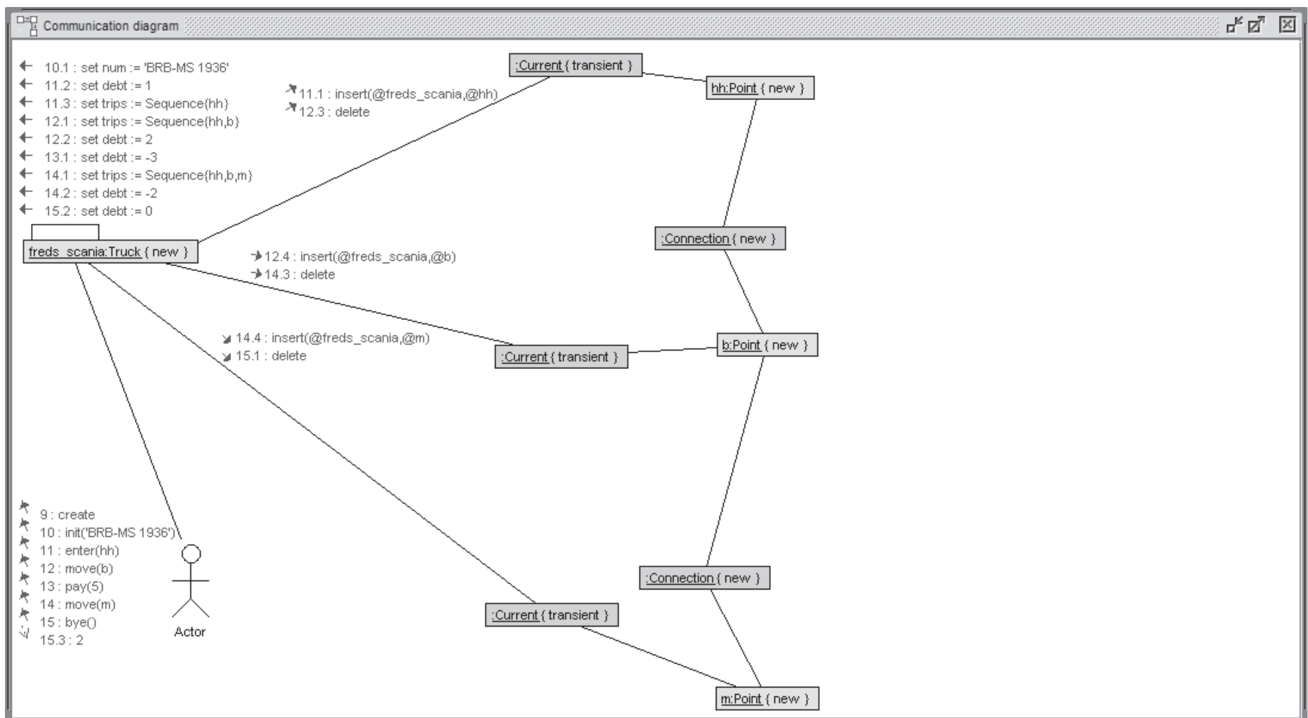

Figure 9: Communication diagram displaying only messages 9-15.

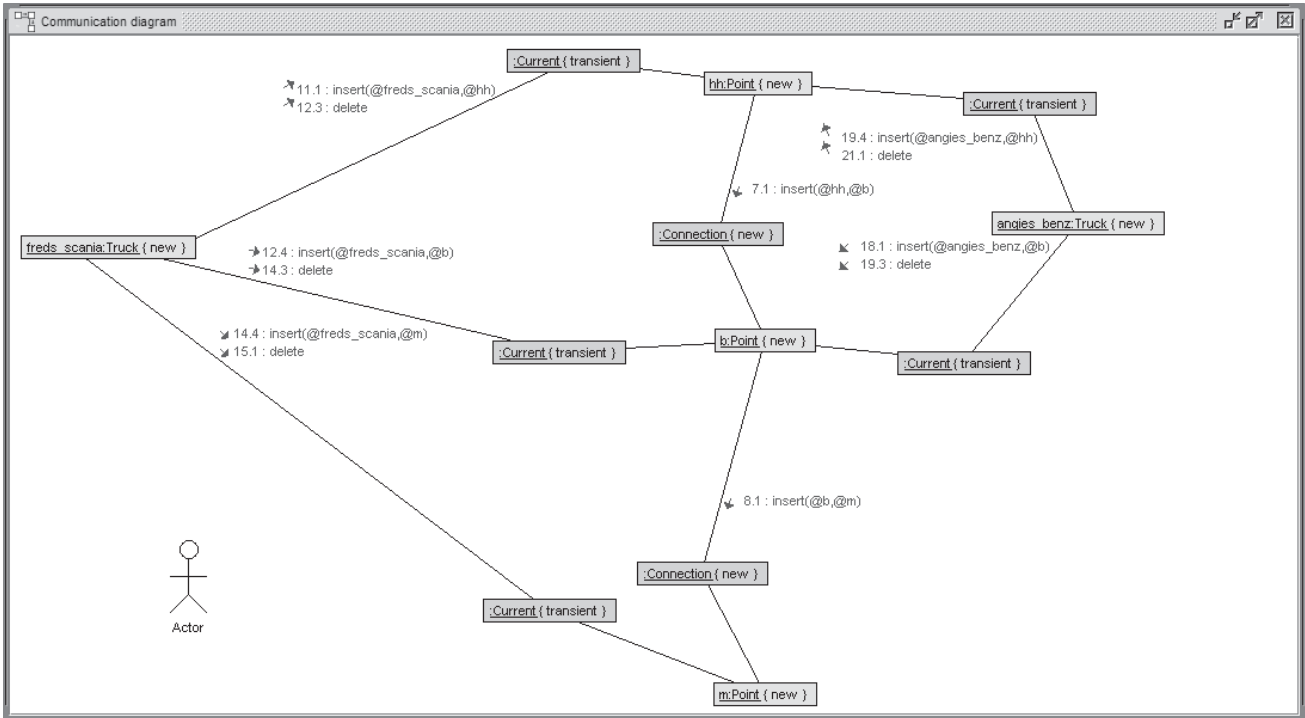

Figure 10: Communication diagram displaying only link insertion and deletion.

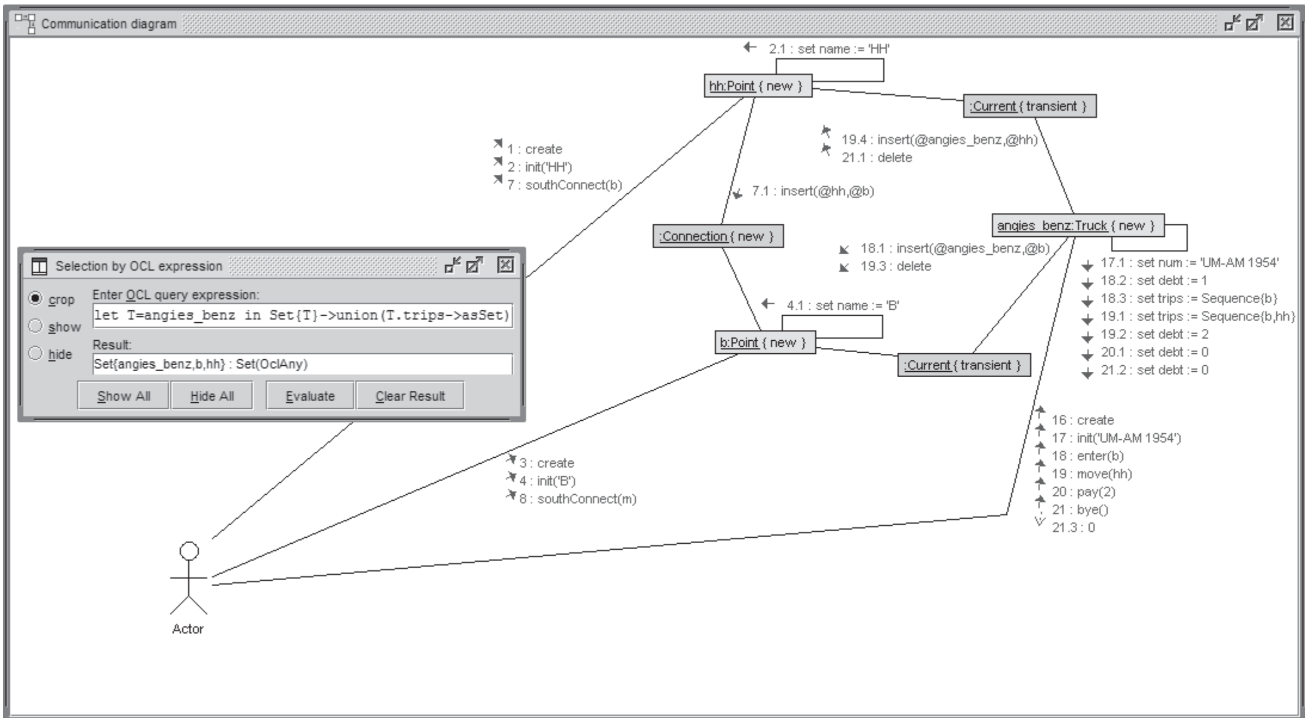

Figure 11: Communication diagram with OCL selection for truck object by identity.

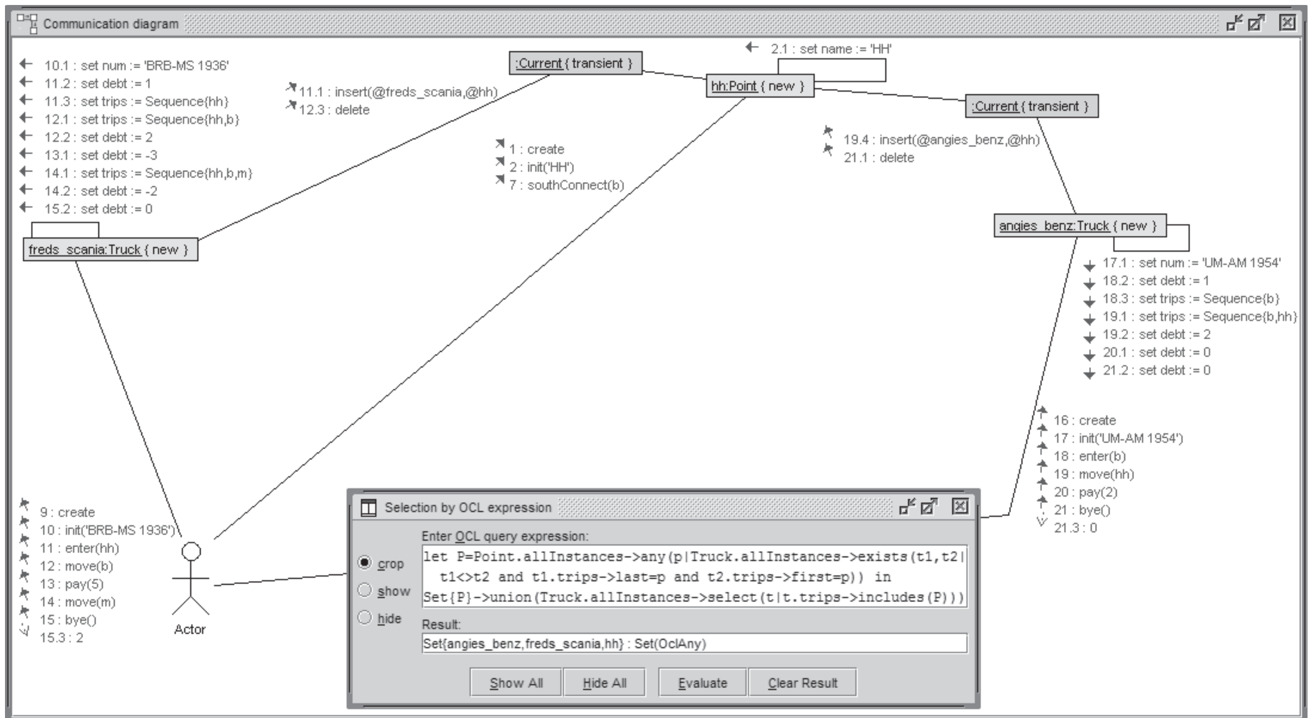

Figure 12: Communication diagram with OCL selection for trucks with coinciding last and first point on trip.

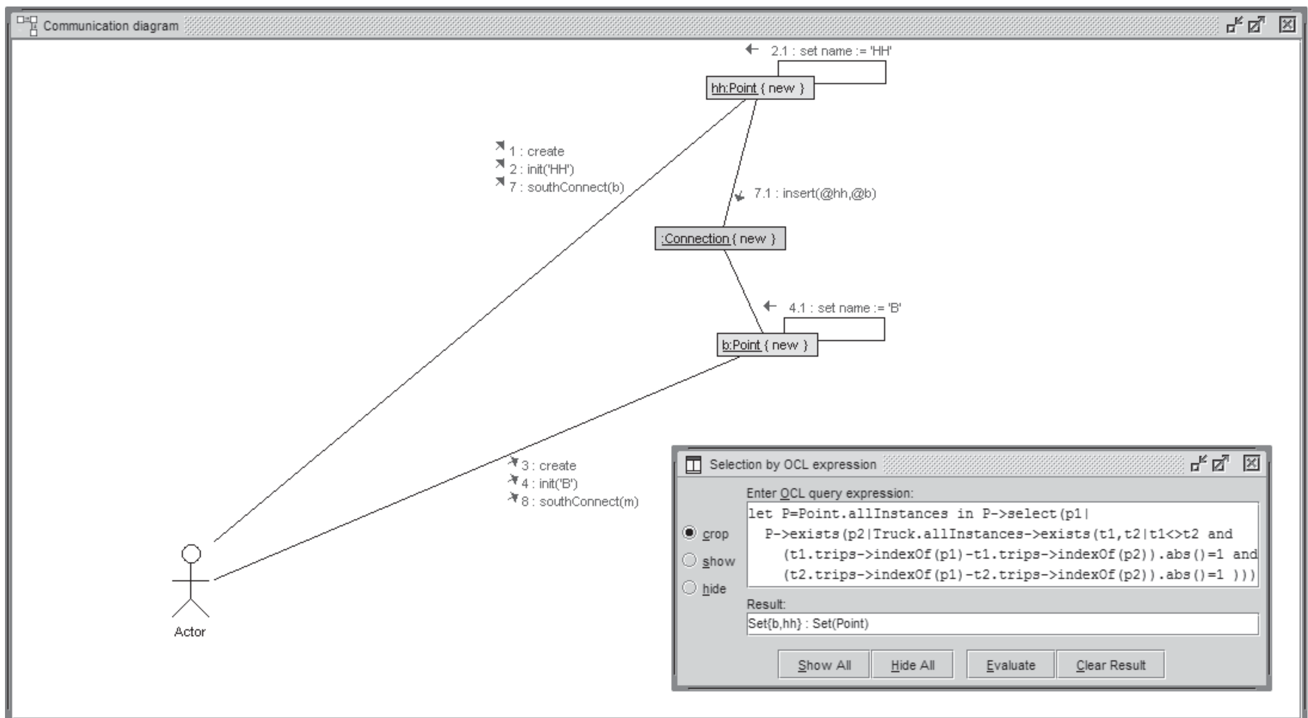

Figure 13: Communication diagram with OCL selection for point connection used twice.

translated into Alloy and shows how to verify aspect interactions with Alloy's model analyzer. In [13], 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 [18]. The approach in [10] 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 [3; 9] 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 [5; 6]) in that we did not consider sequence diagrams with statechart states on lifelines or communication diagrams at all.

9. 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 a desirable feature set 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 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 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.

[2] D. Calegari, M. V. Cengarle, and N. Szasz. UML 2.0 Interactions with OCL/RT Constraints. In *FDL*, pages 167–172. IEEE, 2008.

[3] H. Y. Chen, C. Li, and T. H. Tse. Transformation of UML Interaction Diagrams into Contract Specifications for Object-oriented Testing. In IEEE [7], pages 1298–1303.

[4] 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`.

[5] 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.

[6] 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.

[7] IEEE, editor. *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Montréal, Canada, 7-10 October 2007*. IEEE, 2007.

[8] 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.

[9] 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 [7], pages 2679–2686.

[10] 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.

[11] Z. Micskei and H. Waeselynck. The Many Meanings of UML2 Sequence Diagrams: A Survey. *Software & Systems Modeling*, 10(4):489–514, 2011.

[12] 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.

[13] A. Nayak and D. Samanta. Model-based Test Cases Synthesis using UML Interaction Diagrams. *ACM SIGSOFT Software Engineering Notes*, 34(2):1–10, 2009.

[14] OMG, editor. *UML Superstructure 2.4.1*. Object Management Group (OMG), Aug. 2011.

[15] OMG, editor. *Object Constraint Language, Version 2.3.1*. OMG, 2012. OMG Document, `www.omg.org`.

[16] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language 2.0 Reference Manual*. Addison-Wesley, Reading, 2003.

[17] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 2003. 2nd Edition.

[18] 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.

[19] 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)=Set{self}->closure(p|p.north)
  southPlus():Set(Point)=Set{self}->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()
--------------------------------------------------------------------
```