Development of UML Descriptions with USE

Martin Gogolla, Mark Richters
University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany
[gogolla|mr]@informatik.uni-bremen.de

Abstract

The Object Constraint Language OCL is part of the Unified Modeling Language UML. Within software engineering, UML is regarded today as an important step towards development of high-quality object-oriented systems. OCL allows to describe system structure by invariants and system behavior by pre- and postconditions. This paper explains the functionality of the UML Specification Environment USE which allows to validate and verify UML and OCL descriptions. The paper also uses a new approach to handle UML statecharts by OCL pre- and postconditions.

1 Introduction

The Unified Modeling Language UML [OMG01, RBJ99] is regarded today as an important standard for the development of software systems. Many commercial tools for UML are available. The Object Constraint Language OCL [OMG01, WK98] is part of standard UML, but up to now OCL is not supported by most commercial tools. OCL is a specification language supporting and enriching the UML with textual details which cannot be expressed in diagramatic form. OCL allows to precisely describe system structure by invariants and system behavior by pre- and postconditions.

Tool support for OCL is beginning to develop. Among the first available tools was our system USE (UML Specification Environment). USE is based on conceptional work on the formal semantics of the OCL and the needed relevant UML features [RG01] and work on the metamodel of OCL [RG99]. The main task of USE is to validate and verify specifications developed within UML and OCL. By validation we mean that the developer can animate the specification by providing a number of test cases and check whether the USE responses meet the intuition. Especially for specifications involving formal aspects we regard such validation systems as extremely helpful because they give feedback to developers in early development stages. By verification we mean, that the test cases are formally checked with respect to invariants and pre- and postconditions. Other tools for OCL include the commercial tool from Boldsoft [Bol02], an OCL compiler [HDF00], the KeY tool [ABB+00], and work supporting development under C++ [Chi01].

USE is now available as version 2.1 and has achieved more and more functionality since it was first introduced. USE is the only OCL tool allowing interactive monitoring of OCL invariants and pre- and postconditions. USE has been successfully applied in various larger case studies(e.g. [ÁES01, GB01]) and in teaching UML. It was employed in submissions for the upcoming new version of UML and has been used for large specifications like the UML metamodel with about 100 classes and hundreds of OCL expressions. USE offers the following functionalities:

1. syntax check for and browsing through textual UML and OCL descriptions,
2. generation of system states through object, attribute and link manipulation,
3. representation of system states as object diagrams,
4. monitoring model inherent and explicit invariants in class diagrams,
5. operation execution and monitoring of pre- and postconditions,
6. representation of operation call sequences as sequence diagrams, and
7. querying system states with OCL expressions.

For realizing these functionalities, the USE system covers central UML language features. The input to USE are UML class diagrams in textual form, OCL invariants, OCL operation descriptions through pre- and postconditions, and UML statecharts which are encoded as OCL pre- and postconditions. USE produces as output UML object and sequence diagrams as well as reports on the validity of invariants and pre- and postconditions in table and text form.

The central idea of the USE tool is to check for software quality criteria like correct functionality of UML descriptions already on the design level in an implementation-independent manner. This approach takes advantage of descriptive design level specifications by expressing properties shorter and in a more abstract way. Such properties are given by invariants and pre- and postconditions, and these are checked by the USE system against the test scenarios, i.e. object diagrams and operation calls given by sequence diagrams, which the user provides. These abstract design level tests are expected to be also used later in the implementation phase.

The rest of this paper is structured as follows. In Sect. 2 we present a small case study which will be used throughout the paper. A small case study was choosen in order to be self-contained, but also to demonstrate the principal idea of specification development with USE. These ideas work for larger descriptions as well. In that case study we also employ a new approach how to translate UML statecharts into OCL pre- and postconditions. The central Sect. 3 explains the functionalities offered by USE through various screenshots. The paper ends with concluding remarks in Sect. 4.

2 Running Example: The Crossing System

Our running example describes a simple traffic control application where a controller is supposed to supervise four traffic lights on a street crossing. Figure 1 shows a UML class diagram with the classes `Controller` and `Light` and two UML statechart diagrams, the left one for class `Controller` and the right one for class `Light`.

A `Controller` object can be linked to the four `Light` objects which are then accessible with the role names `north`, `south`, `west`, and `east`. The controller possesses an operation `createLights` for creating and initializing the lights and an operation `switch` for

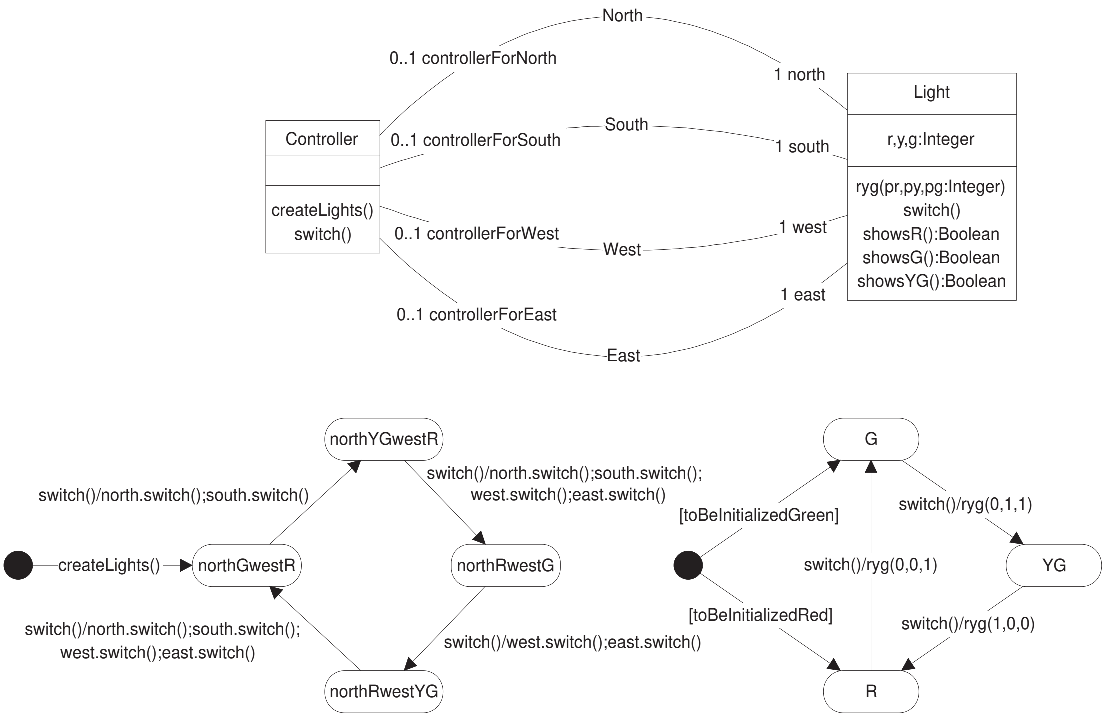

Figure 1: Class Diagram and Statecharts for Crossing System

switching the crossing system. The controller is responsible for transmitting a `switch` operation to the respective lights. A `Light` object possesses three attributes `r`, `y`, and `g` for the red, yellow, and green bulb, respectively, an internal operation `ryg` for manipulating the three attributes and also an operation `switch` for switching. Furthermore, for convenience, it has three Boolean-valued operations for checking the displayed signal.

The `Light` statechart shows that we describe Italian traffic lights with three states `G`, `YG`, and `R`. In state `G` only the green bulb is on, in state `YG` only the yellow and green bulb, and in state `R` only the red bulb. The shows predicates characterize these states. The `switch` operation in class `Light` switches between these states in a cyclic way. The `Controller` statechart has four states which are named according to the signal on the `north` and `west` light. We require that the `north` and `south` lights coincide as well as the `west` and `east` lights. When receiving a `switch` operation, the task of the controller is to transmit the `switch` to the proper `Light` objects. Certain properties have to be satisfied. For example, for safety reasons, the `north` and the `west` light are not allowed to display only green at the same time, and they are not allowed to show only red at the same time. The purpose of USE is to bring life into this description.

3 Use of USE

We now explain how a UML description like the one in Fig. 1 can be analyzed with the USE tool. We do so by showing typical screenshots taken from sessions working with USE. But before going into details we want to give an overview on the content of the displayed screenshots in Figs. 2, 3 and 4.

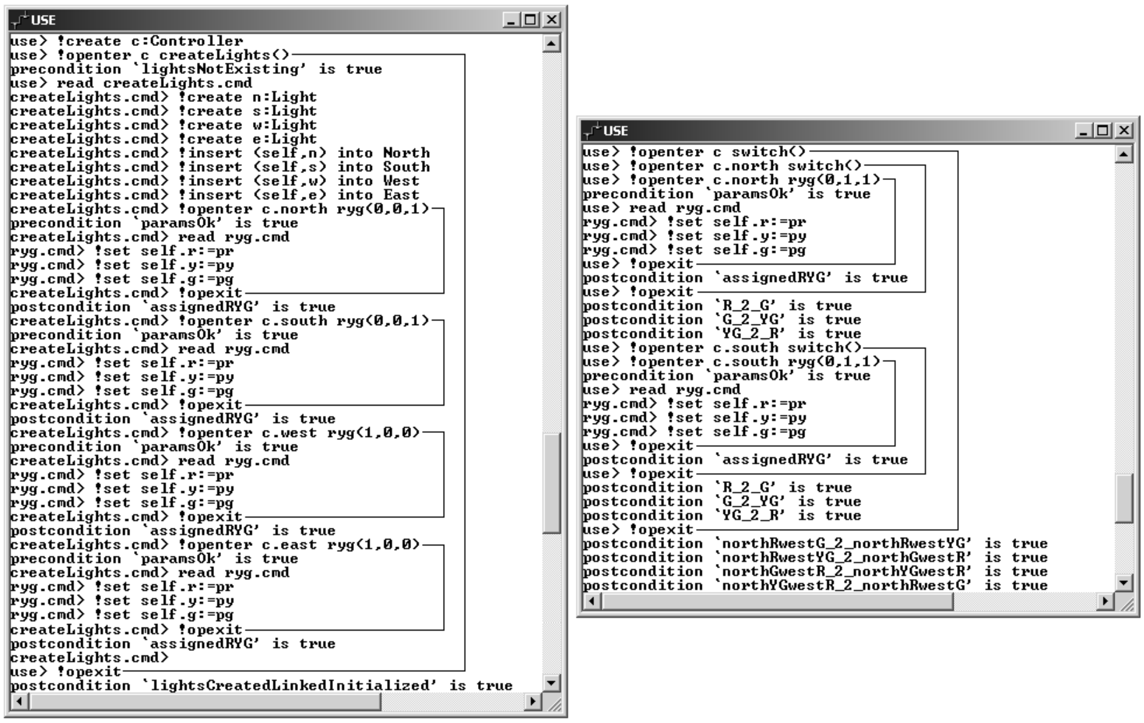

Figure 2: USE Command Windows

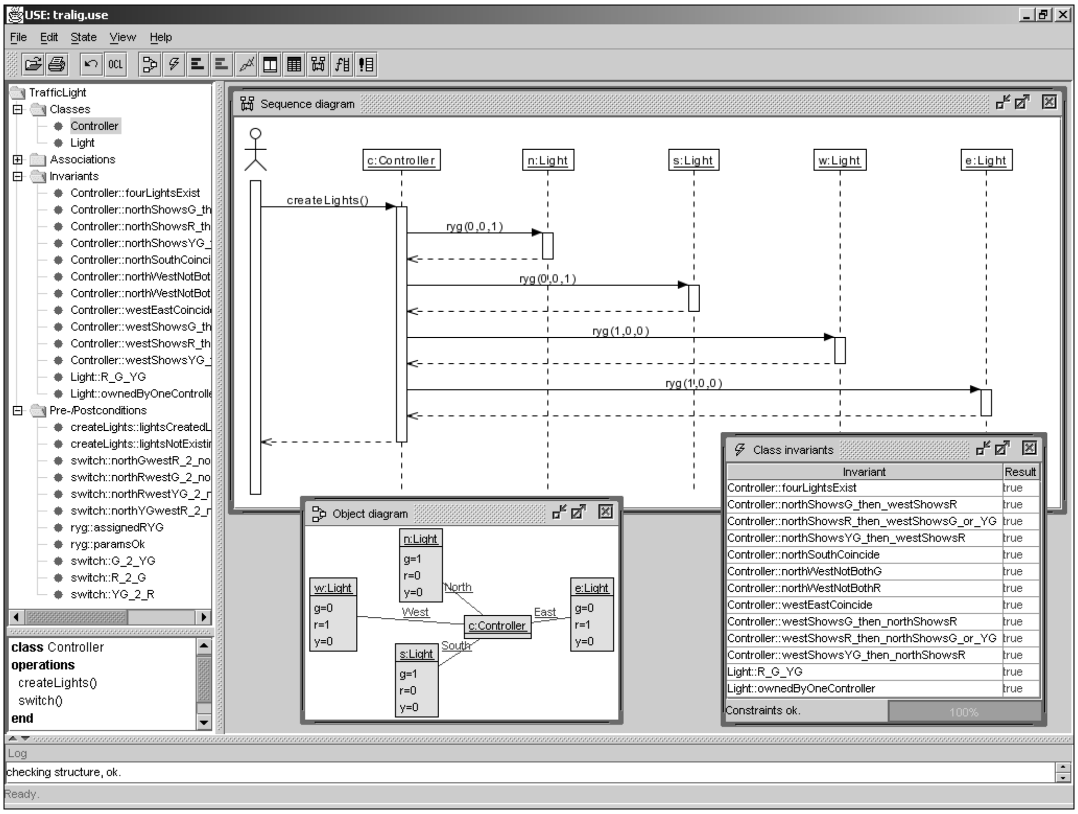

Figure 3: USE Project and Snapshot Windows - Initialization

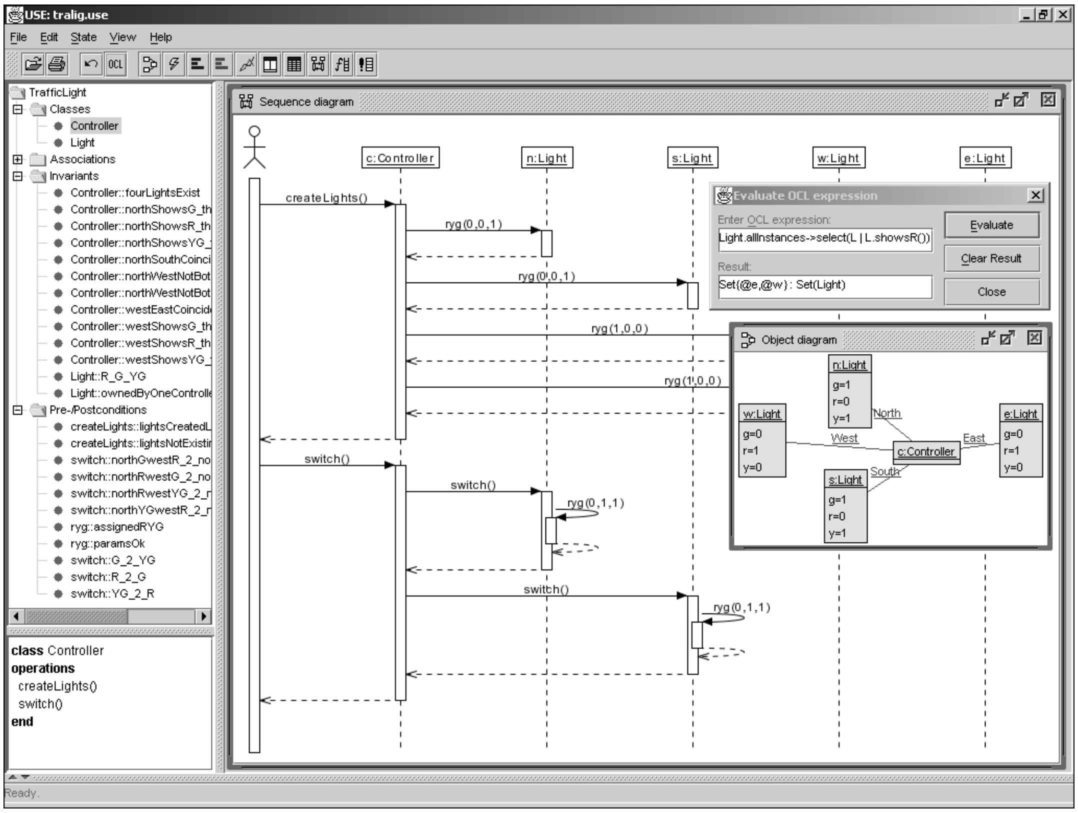

Figure 4: USE Project and Snapshot Windows - Switching

The starting point for working with USE is to load a USE specification like the one given in [GR02]. After successful syntactical analysis, the specification is available in a project browser window as shown in the left of Fig. 3. By selecting an element of the specification, like for example the class `Controller`, this element is shown in detail in a window below the project browser window. In Fig. 3, the details of class `Controller` are pictured.

In Fig. 2 we see commands which are given to the USE system and the responses of the USE system. Roughly speaking, the commands in the left part of Fig. 2 generate the situation in Fig. 3, and the commands in the right part of Fig. 2 subsequently generate the situation in Fig. 4. Such commands construct system states which can be inspected through various windows as shown in Figs. 3 and 4. On the bottom of Fig. 3, a log window gives information on the status of the model inherent constraints, namely whether the specified multiplicities are currently valid or not. To the right of the project browser window, we see three further windows: A sequence diagram window, an object diagram window, and a class invariant window. The sequence diagram window shows which operations have been applied to which object and displays the operation order. The object diagram window shows the current system state, i.e. the objects currently existing and their attributes as well as their links. The class invariant window shows the value of the specified invariants, i.e. it reports on whether the respective invariant is false or true.

Figure 4 shows in the sequence diagram the development after some more commands have been entered. The object diagram in Fig. 4 shows the modified system state.

Figure 4 also explains through the OCL evaluation window shown in the upper right part how system states can be queried with OCL expressions. Let us now go through the USE functionalities as mentioned in the introduction and discuss how they are realized in USE.

3.1 Syntax Check

UML class diagrams, invariants, pre- and postconditions as well as UML statecharts are given to USE in textual form. [GR02] shows the complete specification for the crossing system.

Class and association descriptions are written down straightforward. Classes possess attributes and operations. Query operations, like for example `showsR`, can be characterized by OCL expressions. Associations (as well as aggregations and compositions) are described with multiplicities and role names.

As usual in OCL, an invariant possesses a class as its context and a name. The crossing system has two invariants for class `Light` and eleven for class `Controller`. For example, the `Light` invariant `R_G_YG` requires that either only red is displayed or only green or only yellow and green. The unwanted case that all three shows predicates become true (this would be allowed by the `xor`) is excluded due to the non-overlapping definition of the shows predicates. Considering an example for a `Controller` invariant, the invariant `northWestNotBothG` disallows system states where both the `north` and `west` lights show green. We do not discuss all invariants in detail but emphasize that a well chosen name for invariants is useful because these names are employed in the invariant window to be discussed further down.

As usual in OCL, operations can be described by pre- and postconditions. The `Controller` operation `createLights` is specified with pre- and postconditions. The precondition requires that the current controller is not linked to any `Light` object. The postcondition ensures that four new `Light` objects are created and linked to the controller and that the `Light` objects are correctly intitialized. The `Light` operation `switch` is characterized by three postconditions which encode the `Light` statechart. The idea is that each transition in the statechart becomes a postcondition where the pre-state of the transition is described by means of the OCL operator `@pre`. The operator `@pre` allows in postconditions access to the values of attributes at the precondition time. Thus for example, the postcondition `R_2_G` is true provided the following holds: If the operation `switch` was called in the state where only red is displayed, then this call of `switch` results in the state where only green is displayed. Therefore the three postconditions for the `Light` operation `switch` realize the `Light` statechart.[1] Analogously to the `Light` statechart, the four postconditions for the `Controller` operation `switch` implement the `Controller` statechart. We refrain from explaining the details of the remaining pre- and postconditions.

[1] Due to the current semantics of pre- and postconditions for operations in UML, which requires that all preconditions have to be true at invocation time, one cannot describe the `Light` statechart more intuitively with three pairs where each pair consists of a precondition and a postcondition.

3.2 System State Generation

System states can be manipulated in USE by entering commands in the command window as shown in Fig. 2. Alternatively such manipulations can also be achieved interactively by using the graphical user interface, but for documentation purposes we here use the command window. The basic manipulation commands are the commands `!create` and `!destroy` for object creation and deletion, the command `!set` for attribute modification, and the commands `!insert` and `!delete` for link modification. Commands can also be read from a command file. In Fig. 2, the two command files `createLights.cmd` and `ryg.cmd` are employed which can be considered as the USE implementation of the respective operation. The content of those command files is as follows.

```
createLights.cmd:                                           ryg.cmd:

!create n:Light             !opexit                         !set self.r:=pr
!create s:Light             !openter c.south ryg(0,0,1)     !set self.y:=py
!create w:Light             read ryg.cmd                    !set self.g:=pg
!create e:Light             !opexit
!insert (self,n) into North !openter c.west ryg(1,0,0)
!insert (self,s) into South read ryg.cmd
!insert (self,w) into West  !opexit
!insert (self,e) into East  !openter c.east ryg(1,0,0)
!openter c.north ryg(0,0,1) read ryg.cmd
read ryg.cmd                !opexit
```

3.3 Object Diagrams

The system state achieved after execution of the commands in the left part of Fig. 2 is displayed by the object diagram in Fig. 3. The four `Light` objects with their attribute values, the one `Controller` object, and their links are shown. Various options influence the way the object diagram is pictured. For example, the display of attribute values and the display of role names on links can be turned on or off.

3.4 Invariants

Invariants in USE are divided into model inherent and explicit ones. The model inherent invariants are given through the UML class diagram with the multiplicities specified therein. The log window in the bottom of Fig. 3 reports that the multiplicity constraints are satisfied by the current system state. The evaluation of the explicit invariants is displayed by the class invariant window. There, all invariant names together with their corresponding classes are pictured. For each invariant the result of the evaluation is given. We see that in the current system state all invariants are valid, and therefore the achieved system state has all desirable properties expressed as invariants. In case of an invariant evaluating to false, double-clicking the invariant would open an evaluation browser window which allows to analyse which parts of the invariant and which objects in the current system state contribute to the undesired result. This will be detailed in Sect. 3.8. As mentioned above, an expressive name for an invariant is important at this point in order to understand the result displayed.

3.5 Pre- and Postconditions

As exemplified in the left part of Fig. 2, operations in USE can be invoked by the command `!openter` and closed by the command `!opexit`. Upon operation invocation the precondition is checked, and after exiting the operation the postcondition is evaluated. For example, after submitting the command line `!openter c createLights()` which starts the respective operation on the `Controller` object `c`, the USE system responds with the message that the precondition of the operation evaluates to true. Thereafter, the implementation of that operation is read from a command file and executed step by step. This implementation also invokes other operations, for example through the call `!openter c.north ryg(0,0,1)`. The invocation `!openter c createLights()` is closed by the last `!opexit` command in the left part of Fig. 2, and the USE system reports at this point that the postcondition `lightsCreatedLinkedInitialized` is satisfied. This means that the chosen implementation of `createLights` meets the desired and specified behavior.

In the right part of Fig. 2 we show how the first call of the `switch` operation on the `Controller` object is handled. After invoking that operation and the induced invocations on the respective `Light` objects connected to the `Controller` object, the last three `!opexit` commands close those invocations, and USE responses that all postconditions of all operations are valid. This means that the implemented behavior through the entered commands is correct with respect to the specification, i.e. the operation calls and their implementation meet the behavior described in the statecharts which were encoded in the pre- and postconditions of the operation `switch`.

3.6 Sequence Diagrams

Sequence diagrams are employed in USE in order to capture complex operation calls. The sequence diagram in Fig. 3 shows that a user object symbolized by the strawman icon invokes the operation `createLights` on the `Controller` object and that the `Controller` object reacts by submitting respective initialization calls to the four `Light` objects. The sequence diagram in Fig. 4 points out that the first `switch` directed to the `Controller` object is transmitted only to the `north` and `south` light and that a `switch` operation on `Light` objects induces a call of the operation `ryg` modifying the bulb attributes. The evaluation of the invariants in this stage would result in the same table as before, i.e. all invariants are valid. Together with the positive results of evaluating the pre- and postconditions presented in Fig. 2, the sequence diagram in Fig. 4 captures the proof that the implemented behavior is correct with respect to specified invariants and the specified pre- and postconditions.

3.7 Querying

Figure 4 also shows in the upper right part how system states can be queried through OCL expressions. The OCL expression evaluator allows to compute the result of complex OCL expressions in the current system state. In the example, all `Light` objects are retrieved which currently display red.

3.8 Further Windows

Figure 5 shows further windows supporting the functionalities for operation invocation and invariant monitoring. The situation shown is an intermediate step reached during execution of the commands given right part of Fig. 2 before the first `!opexit` is issued. The sequence diagram in Fig. 5 displays this fact because the dashed return arrows present in Fig. 4 do not yet appear here. The fact, that three operation calls are not yet closed, is reflected in the call stack window in the middle of Fig. 5 which exactly shows these three open operation calls.

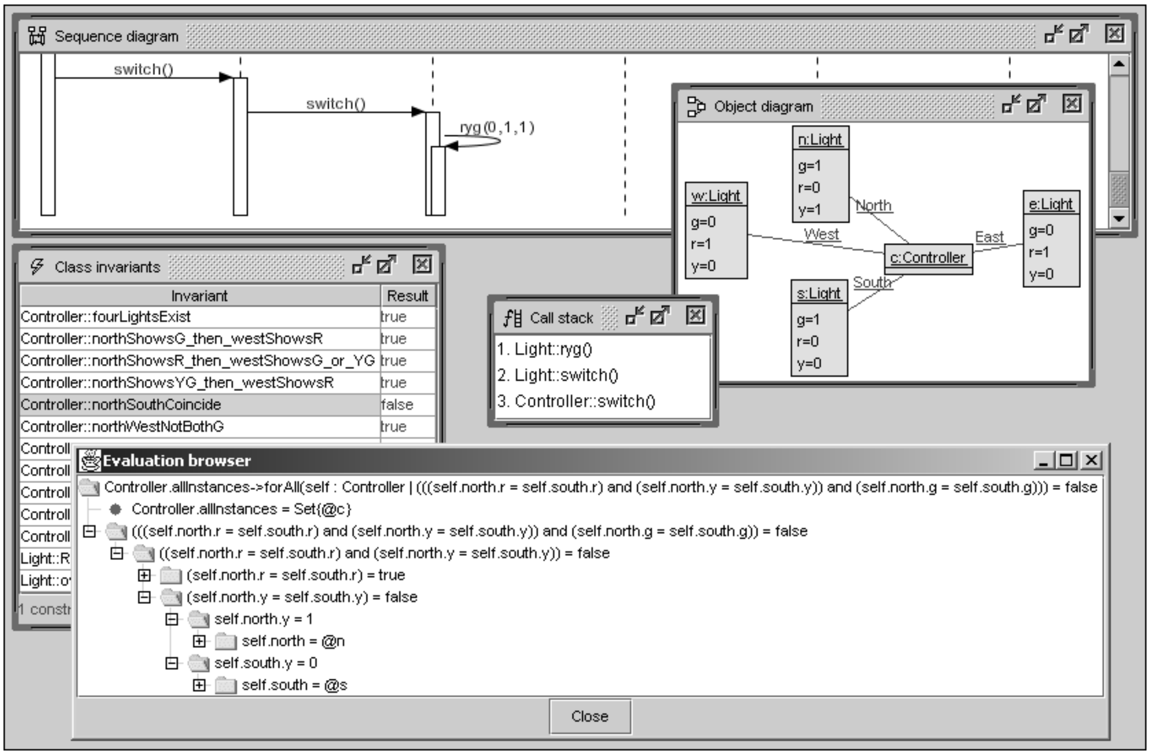

Figure 5: USE Call Stack and Evaluation Browser Windows

At this point, the class invariant window in the left part also shows that the invariant `northSouthCoincide` evaluates to false. Double-clicking that invariant in the class invariant window opens the evaluation browser window shown in the bottom of Fig. 5. That window traces which parts of the invariant and which part of the current system state contribute to the invariant evaluating to false. That window reveals that the `Light` objects `n` and `s` are responsible for the fact that the invariant's subexpression `north.y=south.y` is false and with this the complete invariant is false. This non-coincidence of the `Light` attribute `y` on objects `n` and `s` is also visually represented by the object diagram.

4 Conclusion

In this paper, we have described the functionality of the UML Specification Environment USE. USE allows to validate formal specifications by providing test cases and checking whether the system's responses meet the intuition. USE also allows to verify formal specifications because it checks whether the operation implementation, namely the chosen operation commands, meet the operation specification. Thus USE covers validation and verification aspects.

Future work will consider the further development of USE. In particular, the results of evaluating pre- and postconditions in the command window could be represented in the graphical user interface. Invoking and finishing operations could also be supported by interactive means. In order to increase the computational power of operations one could consider programming language-like features, i.e. conditionals and loops, for the command interpretation.

References

[ABB+00] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I.P. de Guzmán, G. Brewka, and L. M. Pereira, editors, *Proc. 8th European Workshop on Logics in AI (JELIA)*, LNCS 1919, pages 21–36. Springer-Verlag, 2000.

[ÁES01] José M. Álvarez, Andy Evans, and Paul Sammut. Mapping between levels in the metamodel architecture. In Martin Gogolla and Cris Kobryn, editors, *Proc. 4th Int. Conf. UML (UML'2001)*, pages 34–46. Springer, Berlin, LNCS 2185, 2001.

[Bol02] Boldsoft. The Boldsoft OCL Tool Model Run. `www.boldsoft.com`, Boldsoft, Stockholm, 2002.

[Chi01] Dan Chiorean. Using OCL Beyond Specifications. In A. Evans, R. France, A. Moreira, and B. Rumpe, editors, *Proc. UML'2001 Workshop on Rigorous Development*, pages 57–68. LNI, German Informatics Society, 2001.

[GB01] Geri Georg and James Bieman. Using Alloy and UML/OCL to Specify Run-Time Configuration Management: A Case Study. In A. Evans, R. France, A. Moreira, and B. Rumpe, editors, *Proc. UML'2001 Workshop on Rigorous Development*, pages 69–70. LNI, German Informatics Society, 2001.

[GR02] Martin Gogolla and Mark Richters. USE Specification Text for the Traffic Light Case Study. `ftp://ftp.informatik.uni-bremen.de/local/db/papers/trali.use`, 2002. *Also shown on the next 2 pages.*

[HDF00] H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000*, LNCS 1939, pages 278–293. Springer, 2000.

[OMG01] OMG, editor. *OMG Unified Modeling Language Specification, Version 1.4*. OMG, September 2001. OMG Document `formal/01-09-68` through `formal/01-09-80`, `www.omg.org`.

[RBJ99] Jim Rumbaugh, Grady Booch, and Ivar Jacobson. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, 1999.

[RG99] Mark Richters and Martin Gogolla. A Metamodel for OCL. In Robert France and Bernhard Rumpe, editors, *Proc. 2nd Int. Conf. Unified Modeling Language (UML'99)*, pages 156–171. Springer, Berlin, LNCS 1723, 1999.

[RG01] Mark Richters and Martin Gogolla. OCL - Syntax, Semantics and Tools. In Tony Clark and Jos Warmer, editors, *Advances in Object Modelling with the OCL*, pages 43–69. Springer, Berlin, LNCS 2263, 2001.

[WK98] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

File ftp://ftp.informatik.uni-bremen.de/local/db/papers/trali.use

```
model TrafficLight
class Controller
operations
  createLights()
  switch()
end
class Light
attributes
  r : Integer
  y : Integer
  g : Integer
operations
  ryg(pr:Integer,py:Integer,pg:Integer)
  switch()
  showsR(): Boolean = (r=1 and y=0 and g=0)
  showsG(): Boolean = (r=0 and y=0 and g=1)
  showsYG():Boolean = (r=0 and y=1 and g=1)
end
association North between
  Controller[0..1] role controllerForNorth
  Light[1] role north
end
association South between
  Controller[0..1] role controllerForSouth
  Light[1] role south
end
association West between
  Controller[0..1] role controllerForWest
  Light[1] role west
end
association East between
  Controller[0..1] role controllerForEast
  Light[1] role east
end
constraints
context Light inv ownedByOneController:
  (controllerForNorth->size)+(controllerForSouth->size)+
  (controllerForWest->size) +(controllerForEast->size)=1
context Light inv R_G_YG:
  showsR() xor showsG() xor showsYG()
context Controller inv fourLightsExist:
  Set{north,south,west,east}->size=4
context Controller inv northSouthCoincide:
  north.r=south.r and north.y=south.y and north.g=south.g
context Controller inv westEastCoincide:
  west.r=east.r and west.y=east.y and west.g=east.g
context Controller inv northWestNotBothG:
  not(north.g=1 and west.g=1)
```

```
context Controller inv northWestNotBothR:
  not(north.r=1 and west.r=1)
context Controller inv northShowsG_then_westShowsR:
  north.showsG() implies west.showsR()
context Controller inv northShowsYG_then_westShowsR:
  north.showsYG() implies west.showsR()
context Controller inv northShowsR_then_westShowsG_or_YG:
  north.showsR() implies (west.showsG() or west.showsYG())
context Controller inv westShowsG_then_northShowsR:
  west.showsG() implies north.showsR()
context Controller inv westShowsYG_then_northShowsR:
  west.showsYG() implies north.showsR()
context Controller inv westShowsR_then_northShowsG_or_YG:
  west.showsR() implies (north.showsG() or north.showsYG())
context Light::switch()
post R_2_G:
  (r@pre=1 and y@pre=0 and g@pre=0) implies showsG()
post G_2_YG:
  (r@pre=0 and y@pre=0 and g@pre=1) implies showsYG()
post YG_2_R:
  (r@pre=0 and y@pre=1 and g@pre=1) implies showsR()
context Light::ryg(pr:Integer,py:Integer,pg:Integer)
pre paramsOk:
  (0<=pr and pr<=1) and (0<=py and py<=1) and (0<=pg and pg<=1)
post assignedRYG:
  r=pr and y=py and g=pg
context Controller::switch()
post northRwestG_2_northRwestYG:
  ( north.r@pre=1 and north.y@pre=0 and north.g@pre=0 and
    west.r@pre=0 and west.y@pre=0 and west.g@pre=1 )
  implies ( north.showsR() and west.showsYG() )
post northRwestYG_2_northGwestR:
  ( north.r@pre=1 and north.y@pre=0 and north.g@pre=0 and
    west.r@pre=0 and west.y@pre=1 and west.g@pre=1 )
  implies ( north.showsG() and west.showsR() )
post northGwestR_2_northYGwestR:
  ( north.r@pre=0 and north.y@pre=0 and north.g@pre=1 and
    west.r@pre=1 and west.y@pre=0 and west.g@pre=0 )
  implies ( north.showsYG() and west.showsR() )
post northYGwestR_2_northRwestG:
  ( north.r@pre=0 and north.y@pre=1 and north.g@pre=1 and
    west.r@pre=1 and west.y@pre=0 and west.g@pre=0 )
  implies ( north.showsR() and west.showsG() )
context Controller::createLights()
pre lightsNotExisting:
  (north->size)+(south->size)+(west->size)+(east->size)=0
post lightsCreatedLinkedInitialized:
  north.oclIsNew and south.oclIsNew and west.oclIsNew and east.oclIsNew and
  north.showsG() and south.showsG() and west.showsR() and east.showsR()
```