USE
A UML based Specification Environment

Preliminary Version 0.1

Database Systems Group
Bremen University

May 24, 2007

Version 0.1

This document represents the first version of the USE documentation. Several section will be completed, added or changed in the following versions (e.g. the USE Generator sections).

Contents

List of Figures

1 Introduction to USE

USE is a system for the specification of information systems. It is based on a subset of the Unified Modeling Language (UML) [Obj99].

1.1 Overview of USE Features

A USE specification contains a textual description of a model using features found in UML class diagrams (classes, associations, etc.). Expressions written in the Object Constraint Language (OCL) are used to specify additional integrity constraints on the model. A model can be animated to validate the specification against non-formal requirements. System states (snapshots of a running system) can be created and manipulated during an animation. For each snapshot the OCL constraints are automatically checked. Information about a system state is given by graphical views. OCL expressions can be entered and evaluated to query detailed information about a system state.

1.2 Working with USE

This section outlines the general workflow for the specification and validation of a UML model. Figure 1.1 gives a general view of the USE approach. Within this section we use an example model specifying the class *Car* including an attribute *mileage* of type *Integer* and an operation *increaseMileage* with one formal parameter an no return value.
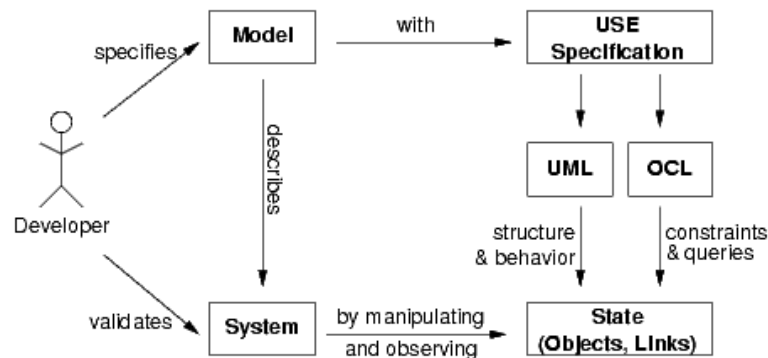
Figure 1.1: Overview of the Specification Workflow

1.2.1 Specifying a UML Model

The USE tool expects a textual description of a model and its constraints as input. (see section 2) The example must therefore be translated into a USE specification[1] by using an external text editor. The USE specification of the example model is shown below.

```
model Cars

class Car
attributes
  mileage : Integer
operations
  increaseMileage(kilometers : Integer)
end
```

1.2.2 Running USE

The following command can be used to invoke USE on the example specification.[2]

```
use ../examples/Documentation/Cars/Cars.use
```

This command will compile and check the file `Cars.use`. There are currently two kinds of user interfaces which can be used simultaneously. The first one is a command line interface where you enter commands at a prompt. The output should be similar to the following.

```
loading properties from: /home/opti/use/etc/use.properties
loading properties from: /home/opti/.userc
use version 2.3.1, Copyright (C) 1999-2006 University of Bremen
compiling specification...
Model Cars (1 class, 0 associations, 0 invariants, 1 operation, 0 pre-/postconditions)
Enter 'help' for a list of available commands.
use>
```

To start USE without loading a specification use the command `use`.

1.2.3 USE Shell - The Command Line Interface

After loading a specification you can enter commands at the prompt. [3] For example, you can enter OCL expressions by starting the input with a question mark. The expression will be evaluated and its result will be shown, e.g.:

```
use> ? Set{1,2,3}->select(e | e > 1)
-> Set{2,3} : Set(Integer)
```

---

[1]A possible extension to USE would be the import of an XMI file created by a CASE tool like Argo UML[4] or Rational Rose

[2]Assuming the current working directory is the top-level directory of the distribution and the bin directory is added to your PATH environment variable.

[3]Try 'help' for a list of available commands.

The command line interface is useful for experienced users and for automated validation procedures since commands can be read from a script file. The graphical user interface is easier to learn and provides different ways of visualizing a system state. By default both interfaces are launched. [1]

1.2.4 Graphical User Interface

The window that appears after starting USE can be seen in the screen shot in figure 1.2. On the left is a tree view showing the contents (classes, associations, invariants, and pre- and postconditions) of the model.
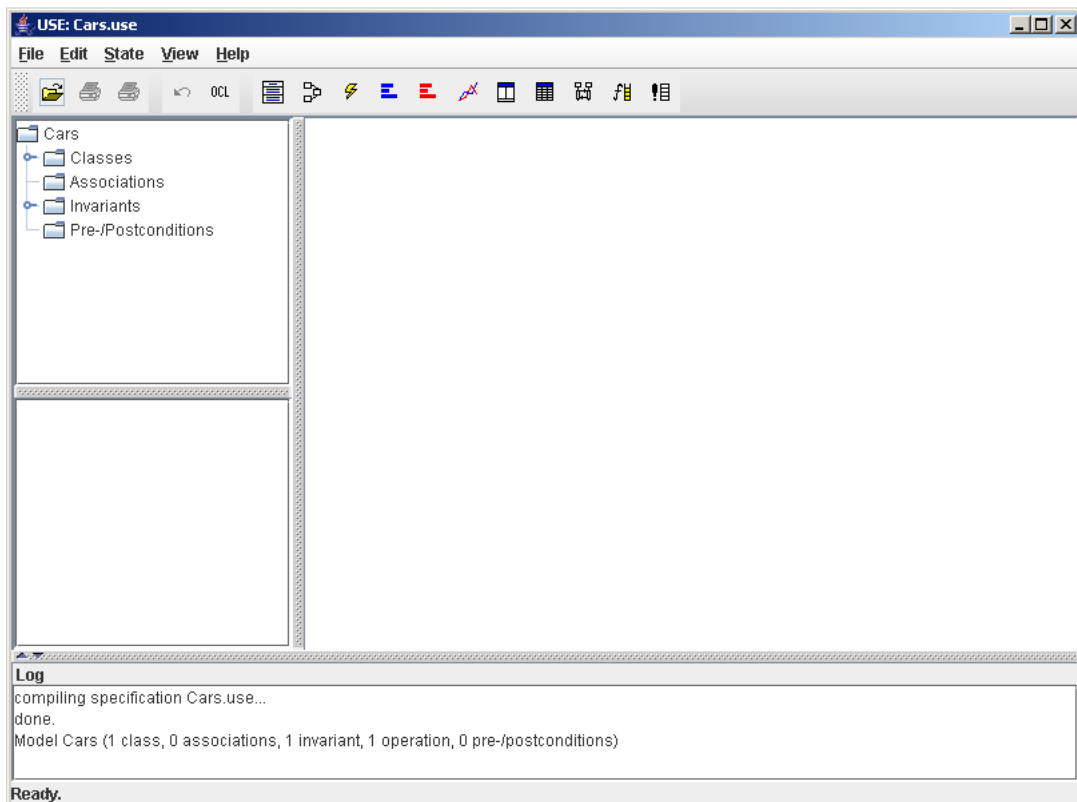
Figure 1.2: Graphical User Interface after starting USE with the Cars Example

The next figure 1.3 shows the expanded tree with all model elements. The invariant is selected and its definition is shown in the panel below the tree.

The large area on the right is a workspace where you can open views visualizing different aspects of a system. Views can be created any time by choosing an entry from the view menu or directly by a toolbar button. There is no limit on active views. The next screenshot shown in

---

[1] Unless you specify the switch -nogui at startup time.
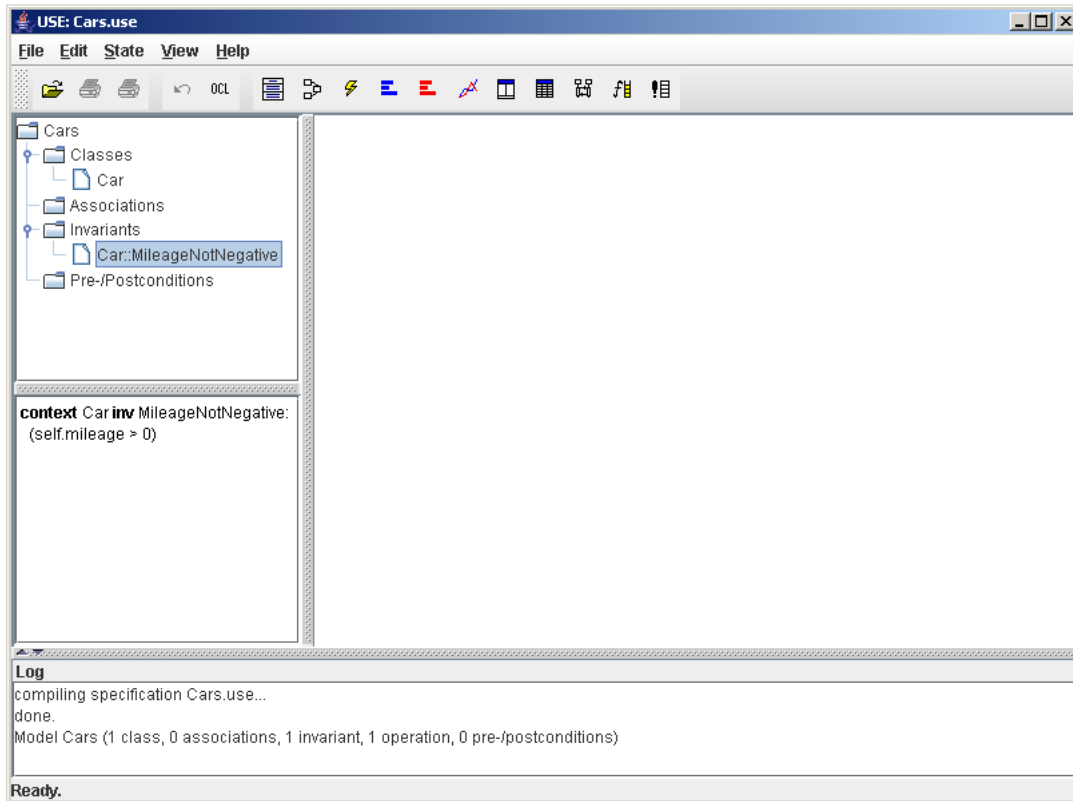
Figure 1.3: Expanded tree with all model elements

figure 1.4 displays the main window after the creation of four views. The two lower views list the names of classes and associations defined in the model and show the number of objects and links in the current system state. The initial system state is empty, i.e., there are no objects and links yet. The view at the upper right displays a list of OCL invariants and their results. As you can see, all invariants are true in the empty system state. Finally, the upper left view will show an object diagram once we have created objects and links.

1.2.5 Creating Objects and Setting Attributes

Now you can create and destroy objects of type *Car* and set their attributes. More complex specifications allow more commands to manipulate the system state. (see section 5.1).
We create two objects *smallCar* and *bigCar* and set their mileage to 2000 resp. −1500 kilometers. The commands are shown below. They can be entered step by step or by reading in a command file. To read in the corresponding command file use the following USE command:
`open ../examples/Documentation/Cars/Cars.cmd`

```
!create smallCar : Car
!create bigCar : Car
```
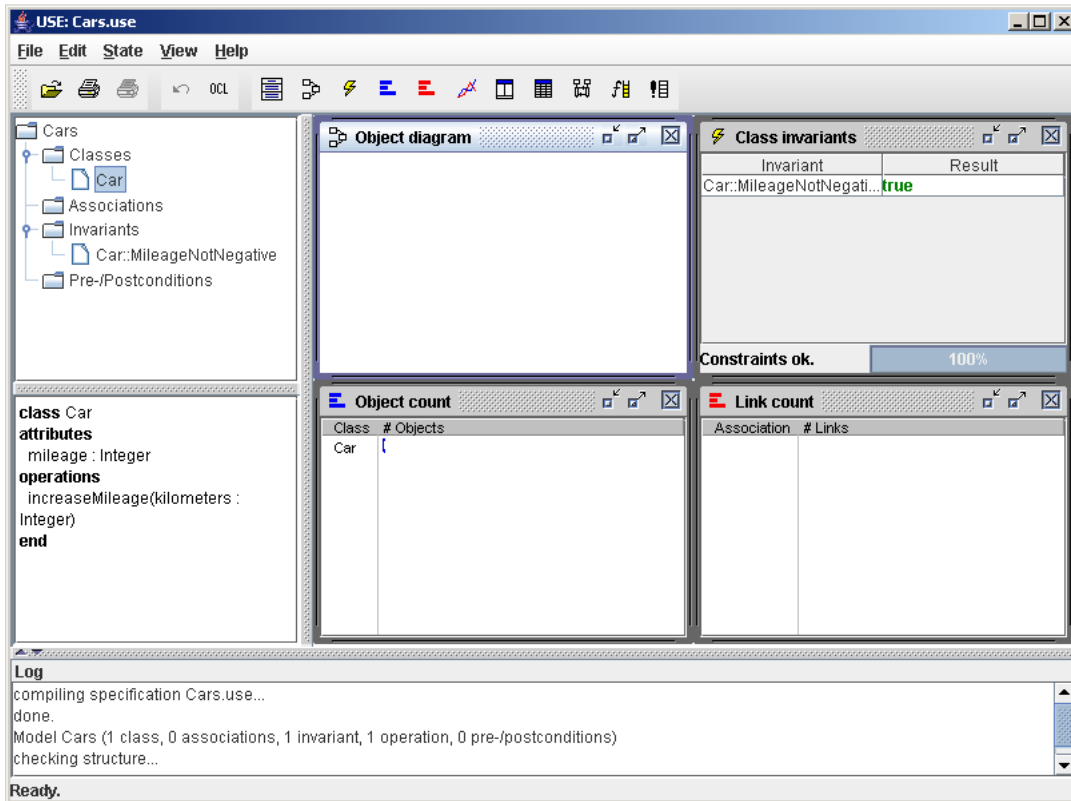
Figure 1.4: Main window with four different views

```
!set smallCar.mileage := 2000
!set bigCar.mileage := -1500
```

1.2.6 Checking OCL Invariants

After creating the system state the Class Invariant View shows that *MileageNotNegative* is violated. (see figure 1.5)
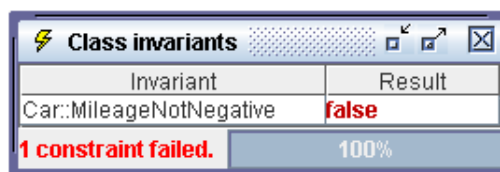
Figure 1.5: Constraint failed

To get more information you can double click on the failed invariant. This opens the Evaluation Browser showing the evaluation of the marked invariant. In figure 1.6 you can see, that object

14

bigCar violates the invariant because its mileage is a negative number.
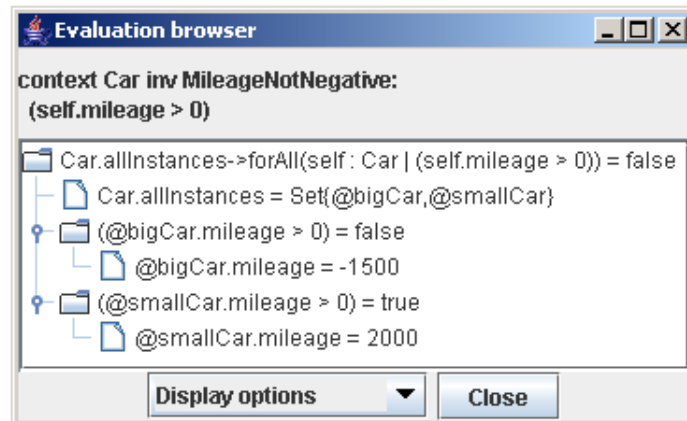
Figure 1.6: Evaluation of the violated invariant

1.2.7 Evaluating OCL Expressions

The OCL parser and interpreter of USE allows the evaluation of arbitrary OCL expressions. The menu item `State|Evaluate OCL expression` opens a dialog where expressions can be entered and evaluated (see figure 1.7).
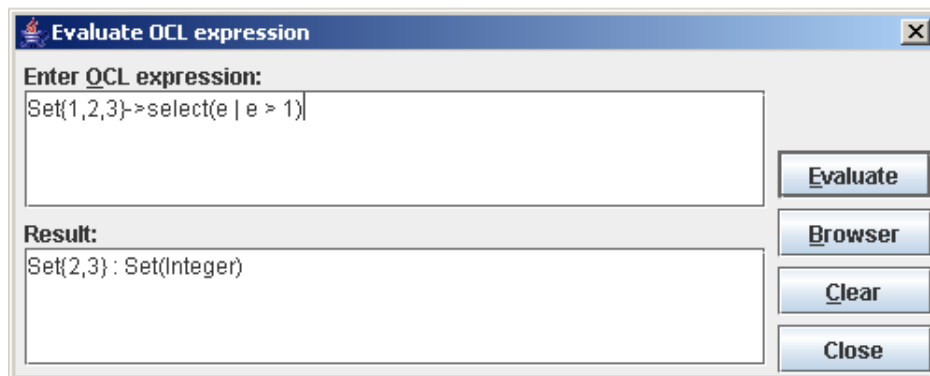
Figure 1.7: Evaluating a simple select expression

The example in figure 1.8 shows a more complex expression with *allInstances* and the collection operations *select*, *collect* and *exists*.

15
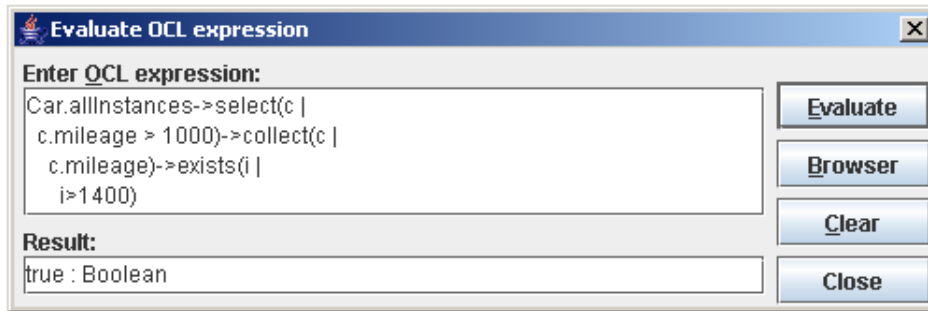
Figure 1.8: Evaluating a more complex expression

1.3 Formal Background

The USE specification language is based on UML and OCL. Due to the semi formal definition of early OCL versions, there were language constructs whose interpretation was ambiguous or unclear [GR98a]. In [GR98b] and [GR99] we have presented a formalization of OCL which was designed to provide a solution for most of the problems and which became part of UML 1.4/1.5. The USE approach to validation is described in [GR00] and [Ric02]. Several other papers of our group employing USE can be found in the publications of our group.[1]

1.4 Examples inspected within this documentation

Beside the cars example there are four examples, which are treated in the course of this documentation.

1.4.1 Employees, Departments and Projects

This example specifies employees working in at least one department. They work on an arbitrary number of projects, which are controlled by exactly one department.
Persons, departments and projects have names identifying them. Departments, which have different locations, and projects have specific budgets. Persons are paid for their job. They have a regular salary. (see figure 1.9)

1.4.2 Persons and Companies

The UML class diagram in figure 1.10 shows an altered model representing persons and companies. Persons have a name, an age, and a salary, which can be raised with the operation *raiseSalary* by a specific amount. They work for at most one company, which has a name and a location. Companies can hire and fire persons.

---

[1]http://www.db.informatik.uni-bremen.de/publications/
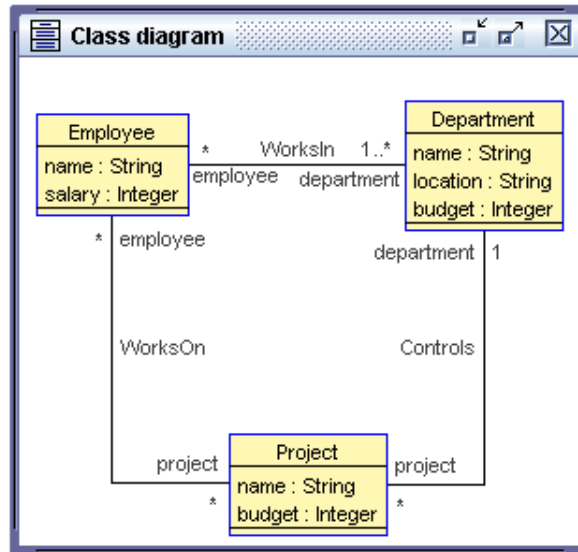
Figure 1.9: Class diagram of the Employees, Departments and Projects example
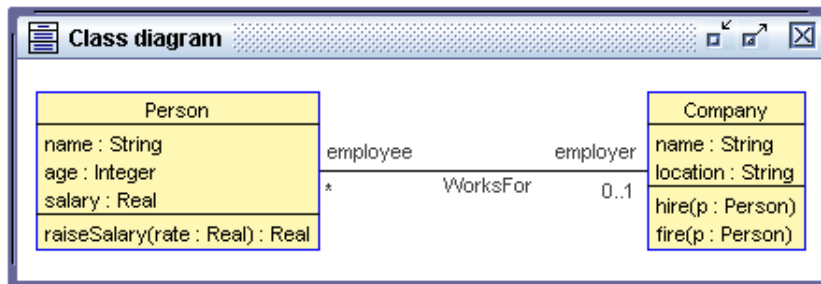
Figure 1.10: Class diagram of the Person & Company example

1.4.3 Graphs

This example is modeling a graph structure. Objects of class *Node* represent nodes of a graph that can be connected by edges. Each node can be connected to an arbitrary number of source and target nodes. The *Node* class contains an operation *newTarget*. The purpose of this operation is to create a new node and to insert a new edge between the source node and the new target node.

1.4.4 Factorial

The factorial example shows how operation calls may be nested in the simulation. It also shows that postconditions may be specified on operations without side effects. An OCL expression can be given to describe the computation of a side effect free operation. In the example, we use a recursive definition of the factorial function. There is only one class *Rec*.
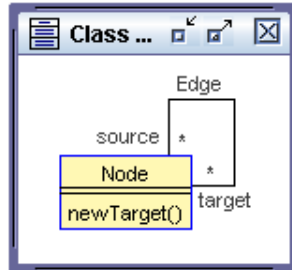
Figure 1.11: Class diagram of the Graph example
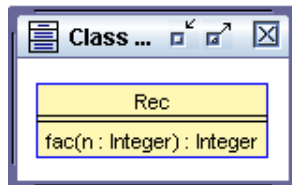
Figure 1.12: Class diagram of factorial example

2 Specifying a UML Model with USE

To define a USE specifications you need an external text editor. The syntactic elements are clarified by a grammar defined with the Extended Backus-Naur Form.

2.1 Defining a UML Model

Every UML Model has a name and an optional body.

Syntax:

$$\langle umlmodel\rangle \quad ::= \quad \texttt{model}\ \langle modelname\rangle\ [\ \langle modelbody\rangle\ ]$$
$$\langle modelname\rangle \quad ::= \quad \langle name\rangle$$

Example: The models name is `Fruits`.

```
model Fruits
...
```

The model body consists of at least one class definition and an arbitrary number of association definitions. Enumeration definitions are allowed at the top of the body. At the end of the specification OCL constraints may be defined.

Syntax:

$$\langle modelbody\rangle \quad ::= \quad \{\ \langle enumerationdefinition\rangle\ \}$$
$$\{\ \langle associationdefinition\rangle\ |\ \langle associationclassdefinition\rangle\ \}$$
$$\langle classdefinition\rangle$$
$$\{\ \langle classdefinition\rangle\ |\ \langle associationdefinition\rangle\ |\ \langle associationclassdefinition\rangle\ \}$$
$$[\ \texttt{constraints}\ \{\ \langle constraintdefinition\rangle\ \}\ ]$$

2.2 Specification Elements

The following sections describe all available elements, which can be used in the model body.

2.2.1 Enumerations

Enumerations may be added at the top of the model body.

Syntax:

$$\langle enumerationdefinition \rangle \quad ::= \quad \texttt{enum} \ \langle enumerationname \rangle \ \{ \ \langle elementname \rangle \ \{ \ , \ \langle elementname \rangle \ \} \ \}$$

$$\langle enumerationname \rangle \quad ::= \quad \langle name \rangle$$

$$\langle elementname \rangle \quad ::= \quad \langle name \rangle$$

Example: An enumeration definition with three elements.

```
enum Flatware {Spoon, Fork, Knife}
```

2.2.2 Classes

A class has a name and optionally attribute and operation definitions. It may be defined as an abstract class.

Syntax:

$$\langle classdefinition \rangle \quad ::= \quad [\ \texttt{abstract}\ ]\ \texttt{class}\ \langle classname \rangle\ [\ \texttt{<}\ \langle classname \rangle\ \{\ ,\ \langle classname \rangle\ \}\ ]$$

$$[\ \texttt{attributes}\ \{\ \langle attributename \rangle : \langle type \rangle\ \}\ ]$$

$$[\ \texttt{operations}\ \{\ \langle operationdeclaration \rangle\ [\ =\ \langle oclexpression \rangle\ ]$$

$$\{\ \langle preconditiondefinition \rangle\ |\ \langle postconditiondefinition \rangle\ \}\ \}\ ]$$

$$[\ \texttt{constraints}\ \{\ \langle invariantdefinition \rangle\ \}\ ]$$

$$\texttt{end}$$

$$\langle classname \rangle \quad ::= \quad \langle name \rangle$$

$$\langle attributename \rangle \quad ::= \quad \langle name \rangle$$

Example: The example shows five different class definitions. The class *Apple* inherits two and the class *Banana* one class. The class *Lemon* is abstract and class *Banana* defines pre- and postconditions for the operation *peel* within its body. The class *Peach* shows how invariants can be integrated into the classes body.

```
class Apple < Orange, Lemon
end

abstract class Orange
attributes
  juice : Boolean
end

class Lemon
```

```
operations
  squeeze(i : Integer) : Integer = i + 1
end

class Banana < Lemon
attributes
  flatware : Set(Sequence(Flatware))
operations
  peel() : String = 'abcd'
      pre: true
      post: 2 = 2
      post: result = 'theResult'
end

class Peach
attributes
operations
constraints
  inv: 3 > 2
  inv neverViolated: true
end
```

2.2.3 Associations

It is possible to define n ary associations . The definition requires a name, at least two partici-
pating classes and multiplicity information. Role names are optional.

Syntax:

⟨*associationdefinition*⟩ ::= (association | composition | aggregation)
⟨*associationname*⟩ between
⟨*classname*⟩ [⟨*multiplicity*⟩] [role ⟨*rolename*⟩] [ordered]
⟨*classname*⟩ [⟨*multiplicity*⟩] [role ⟨*rolename*⟩] [ordered]
{ ⟨*classname*⟩ [⟨*multiplicity*⟩] [role ⟨*rolename*⟩] [ordered] }
end

⟨*multiplicity*⟩ ::= (* | ⟨*digit*⟩ { ⟨*digit*⟩ } [.. (* | ⟨*digit*⟩ { ⟨*digit*⟩ })])
{ , (* | ⟨*digit*⟩ { ⟨*digit*⟩ } [.. (* | ⟨*digit*⟩ { ⟨*digit*⟩ })]) }

⟨*associationname*⟩ ::= ⟨*name*⟩

⟨*rolename*⟩ ::= ⟨*name*⟩

Example: This Examples shows a binary and a tertiary association with different multiplicities
and optional role names. The first association is defined as a composition. The diamond

appears at the first participating class. You may order the elements by using the keyword
`ordered`.

```
composition AppleSpritzer between
  Apple[*] role base
  Lemon[1..8,10,15..*] role flavor
end
```

```
association Ingredients between
  Apple[*] role mainIngredient
  Orange[1]
  Lemon[1..*] role lemon ordered
end
```

2.2.4 Association classes

Association classes combine the body elements of classes and associations.

Syntax:

$\langle associationclassdefinition \rangle$ ::= [abstract] associationclass $\langle classname \rangle$
[< $\langle classname \rangle$ { , $\langle classname \rangle$ }] between
$\langle classname \rangle$ [$\langle multiplicity \rangle$] [role $\langle rolename \rangle$] [ordered]
$\langle classname \rangle$ [$\langle multiplicity \rangle$] [role $\langle rolename \rangle$] [ordered]
{ $\langle classname \rangle$ [$\langle multiplicity \rangle$] [role $\langle rolename \rangle$] [ordered] }
[attributes { $\langle attributename \rangle$: $\langle type \rangle$ }]
[operations { $\langle operationdeclaration \rangle$ [= $\langle oclexpression \rangle$]
{ $\langle preconditiondefinition \rangle$ | $\langle postconditiondefinition \rangle$ } }]
[constraints { $\langle invariantdefinition \rangle$ }]
end

Example: The example defines an association class between two classes. It has two attributes
and one operation, which is no query.

```
associationclass FruitSalad < Orange
 between
  Banana[0..1]
  Apple[1..*]
attributes
  name : String
  weight : Real
operations
  putIn(apple : Apple, banana : Banana)
end
```

22

2.2.5 Constraints

The keyword `constraints` indicates the begin of constraint definition segment. An arbitrary number of invariants may be defined in context of a class. In addition to that, you may define pre- and postconditions to constrain operations. Every constraint can be named.

Syntax:

⟨*constraintdefinition*⟩	::=	⟨*invariantcontext*⟩ | ⟨*operationcontext*⟩
⟨*invariantcontext*⟩	::=	`context` [⟨*variablename*⟩ `:`] ⟨*classname*⟩
		{ ⟨*invariantdefinition*⟩ }
⟨*operationcontext*⟩	::=	`context` ⟨*classname*⟩ ⟨*operationconstraints*⟩
⟨*invariantdefinition*⟩	::=	`inv` [⟨*invariantname*⟩] `:` ⟨*booleanoclexpression*⟩
⟨*operationconstraints*⟩	::=	`::` ⟨*operationdeclaration*⟩
		(⟨*preconditiondefinition*⟩ | ⟨*postconditiondefinition*⟩)
		{ ⟨*preconditiondefinition*⟩ | ⟨*postconditiondefinition*⟩ }
⟨*preconditiondefinition*⟩	::=	`pre` [⟨*preconditionname*⟩] `:` ⟨*booleanoclexpression*⟩
⟨*postconditiondefinition*⟩	::=	`post` [⟨*postconditionname*⟩] `:` ⟨*booleanoclexpression*⟩)
⟨*invariantname*⟩	::=	⟨*name*⟩
⟨*variablename*⟩	::=	⟨*name*⟩
⟨*preconditionname*⟩	::=	⟨*name*⟩
⟨*postconditionname*⟩	::=	⟨*name*⟩

Example: The first two definitions are showing three invariants of the class *Orange*. The second definition defines the variable *orange* which may be used in the OCL expression similar to *self*. The third invariant is not named. USE will assign a name like *inv2* to it. Two preconditions and one postcondition constrain the Operation *squeeze* in class *Lemon*.

```
context Orange
 inv OrangeInv: 1 = 1

context orange : Orange
 inv alwaysTrue: orange = orange
 inv: juice = true

context Lemon :: squeeze(i : Integer) : Integer
   pre: i>0
   pre lessThanTenOranges: i<10
   post alwaysTrue: true
```

2.2.6 Operation declarations

The declaration of an operation consists of the operation name, optional parameters and the return type.

Syntax:

$$\langle\textit{operationdeclaration}\rangle \quad ::= \quad \langle\textit{operationname}\rangle \; (\; [\; \langle\textit{parameters}\rangle \;]\;) \; [\; : \langle\textit{type}\rangle \;]$$

$$\langle\textit{parameters}\rangle \quad ::= \quad \langle\textit{parametername}\rangle : \langle\textit{type}\rangle \; \{\; , \langle\textit{parametername}\rangle : \langle\textit{type}\rangle \;\}$$

$$\langle\textit{operationname}\rangle \quad ::= \quad \langle\textit{name}\rangle$$

$$\langle\textit{parametername}\rangle \quad ::= \quad \langle\textit{name}\rangle$$

Example: Three operation declarations are shown within the class definition example.

2.2.7 Types

Types may be simple ($\langle\textit{simpletype}\rangle$) or complex ($\langle\textit{collectiontype}\rangle$).

Syntax:

$$\langle\textit{type}\rangle \quad ::= \quad \langle\textit{collectiontype}\rangle \,|\, \langle\textit{simpletype}\rangle \,|\, \langle\textit{enumerationname}\rangle$$

$$\langle\textit{collectiontype}\rangle \quad ::= \quad (\; \texttt{Set} \,|\, \texttt{Bag} \,|\, \texttt{Sequence} \;) \;($$
$$\{\; \langle\textit{collectiontype}\rangle \,|\, \langle\textit{simpletype}\rangle \,|\, \langle\textit{enumerationname}\rangle \;\} \;)$$

$$\langle\textit{simpletype}\rangle \quad ::= \quad \texttt{Integer} \,|\, \texttt{Real} \,|\, \texttt{Boolean} \,|\, \texttt{String} \,|\, \langle\textit{classname}\rangle$$

Example: The attribute *flatware* has a complex type.

2.2.8 Names, Numbers and OCL-Expressions

$$\langle\textit{name}\rangle \quad ::= \quad (\; \langle\textit{letter}\rangle \,|\, \_ \;) \; \{\; \langle\textit{letter}\rangle \,|\, \langle\textit{digit}\rangle \,|\, \_ \;\}$$

$$\langle\textit{letter}\rangle \quad ::= \quad \texttt{a} \,|\, \texttt{b} \,|\, \ldots \,|\, \texttt{z} \,|\, \texttt{A} \,|\, \texttt{B} \,|\, \ldots \,|\, \texttt{Z}$$

$$\langle\textit{digit}\rangle \quad ::= \quad \texttt{0} \,|\, \texttt{1} \,|\, \ldots \,|\, \texttt{9}$$

$$\langle\textit{oclexpression}\rangle \quad ::= \quad (^{*} \text{ Replace this symbol by an ordinary OCL expression. } ^{*})$$

$$\langle\textit{booleanoclexpression}\rangle \quad ::= \quad (^{*} \text{ Replace this symbol by an ordinary OCLexpression}$$
$$\text{which results in a boolean value. } ^{*})$$

2.3 Specifications of the Examples

The USE specification of the example models shown in section 1.4 are presented in this section.

2.3.1 Employees, Departments and Projects

The first part of the specification shown below describes the structural information of the class diagram. There are classes with attributes and associations with different multiplicities.

```
model Company

-- classes

class Employee
attributes
  name : String
  salary : Integer
end

class Department
attributes
  name : String
  location : String
  budget : Integer
end

class Project
attributes
  name : String
  budget : Integer
end

-- associations

association WorksIn between
  Employee[*]
  Department[1..*]
end

association WorksOn between
  Employee[*]
  Project[*]
end

association Controls between
  Department[1]
  Project[*]
end
```

We extend the model by the following four constraints which place further restrictions on systems conforming to the model. The constraints are first given in natural language and will later be expressed more formally in OCL (Object Constraint Language). Constraints:

1. The number of employees working in a department must be greater or equal to the number

of projects controlled by the department.

2. Employees get a higher salary when they work on more projects.

3. The budget of a project must not exceed the budget of the controlling department.

4. Employees working on a project must also work in the controlling department.

The goal of applying the USE tool is to interactively validate the above model and the constraints. Objects and links can be created which constitute a system state reflecting a snapshot of a running system. In every system state, the constraints are automatically checked for validity.

In the second part of the specification, we define the constraints in OCL. Each constraint is defined as an invariant in context of a class.

```
-- OCL constraints

constraints

context Department
    -- the number of employees working in a department must
    -- be greater or equal to the number of projects
    -- controlled by the department
  inv MoreEmployeesThanProjects:
    self.employee->size >= self.project->size

context Employee
    -- employees get a higher salary when they work on
    -- more projects
  inv MoreProjectsHigherSalary:
    Employee.allInstances->forAll(e1, e2 |
      e1.project->size > e2.project->size
        implies e1.salary > e2.salary)

context Project
    -- the budget of a project must not exceed the
    -- budget of the controlling department
  inv BudgetWithinDepartmentBudget:
    self.budget <= self.department.budget


    -- employees working on a project must also work in the
    -- controlling department
  inv EmployeesInControllingDepartment:
    self.department.employee->includesAll(self.employee)
```

The complete specification is also available in the file Demo.use[1] in the example directory of the distribution.

2.3.2 Persons and Companies

Here is the USE specification of the class diagram shown in figure 1.10.

```
model Employee

-- classes

class Person
attributes
  name : String
  age : Integer
  salary : Real
operations
  raiseSalary(rate : Real) : Real
end

class Company
attributes
  name : String
  location : String
operations
  hire(p : Person)
  fire(p : Person)
end

-- associations

association WorksFor between
  Person[*] role employee
  Company[0..1] role employer
end
```

We add pre- and postconditions for the *hire* and *fire* operations in class Company. The USE specification is extended as follows.

```
-- constraints

constraints

context Company::hire(p : Person)
```

---

[1]http://www.db.informatik.uni-bremen.de/projects/USE/Demo.use

```
  pre  hirePre1: p.isDefined()
  pre  hirePre2: employee->excludes(p)
  post hirePost: employee->includes(p)

context Company::fire(p : Person)
  pre  firePre:  employee->includes(p)
  post firePost: employee->excludes(p)
```

The first precondition of the hire operation is named *hirePre1* and makes sure that the operation can only be called with a well defined person object.[1] The second precondition *hirePre2* makes sure that the person passed as parameter *p* is not already an employee of the company. The postcondition *hirePost* guarantees that after the operation has exited, the person actually has been added to the set of employees. The constraints for the operation *fire* work just the other way round.

2.3.3 Graphs

The USE specification of the graphs example (1.4.3) is shown below.

```
model Graph

class Node
operations
  newTarget()
end

association Edge between
  Node[*] role source
  Node[*] role target
end

constraints

context Node::newTarget()
  -- the operation must link exactly one target node
  post oneNewTarget:
    (target - target@pre)->size() = 1

  -- the target node must not exist before
  post targetNodeIsNew:
    (target - target@pre)->forAll(n | n.oclIsNew())
```

The postcondition *targetNodeIsNew* also demonstrates the application of the OCL operation *oclIsNew* to check for the creation of new objects.

---

[1]Note that the operation *isDefined* is a USE extension. It is not possible to express this constraint with standard OCL.

2.3.4 Factorial

The factorial example presented in section 1.4.4 is specified below.

```
model NestedOperationCalls

class Rec
operations
  fac(n : Integer) : Integer =
    if n <= 1 then 1 else n * self.fac(n - 1) endif
  end

constraints

context Rec::fac(n : Integer) : Integer
  pre:  n > 0
  post: result = n * fac(n - 1)
```

The postcondition reflects the inductive case of the definition of the factorial function.

3 Analyzing the formal Specification

After specifying a UML model within a `.use` file you can open it with USE. The USE system will parse and type check the file automatically. Possible errors are listed in the log window. (see section 4.3.4)

3.1 Creating System States

The Employees, Departments and Projects example specified in section 2.3.1 is used to show how system states can be created and invariants evaluated.

Objects can be created by selecting a class and specifying a name for the object. The menu command `State|Create object` opens a dialog where this information can be entered (shown in figure 3.1).
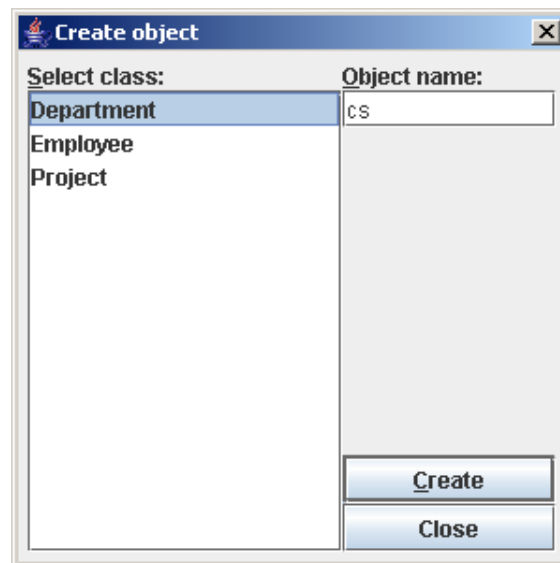
Figure 3.1: Create object dialog

Alternatively, the following command can be used at the shell to achieve the same effect.

```
use> !create cs:Department
```

And, even simpler, an object can be created via drag & drop. Just select a class in the model browser (see section 4.3.3) and drag it to the object diagram.

Figure 3.2 is similar to figure 1.4, but the specification changed to the Employees, Departments and Projects example and the object *cs* was created. The lower left view indicates that there is now one *Department* object, and the object diagram shows this object graphically.
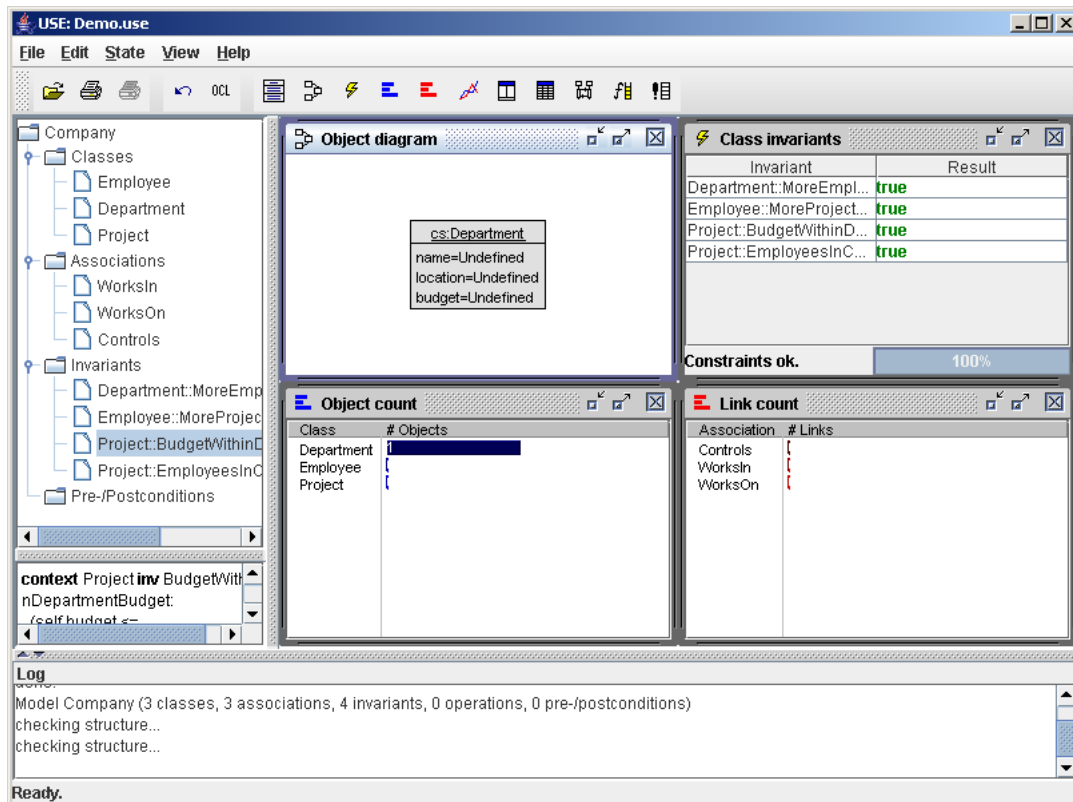
Figure 3.2: Main window with views after creating a new object

A context menu available on a right mouse click in the object diagram provides several display options. For example, the automatic layout can be turned off, the layout of the diagram can be saved and restored from a file, etc. In the previous picture we have turned on the display of attribute values. You can see that the attribute values of the *Department* object are all undefined. For changing attribute values, we can use the `set` command:

```
use> !set cs.name := 'Computer Science'
use> !set cs.location := 'Bremen'
use> !set cs.budget := 10000
```

Attributes can also be changed with an Object Properties View. If you choose `View|Create|Object Properties` from the `View` menu and select the *cs* object, you get the view shown in figure 3.3 where you can inspect and change attributes of the selected object.

31
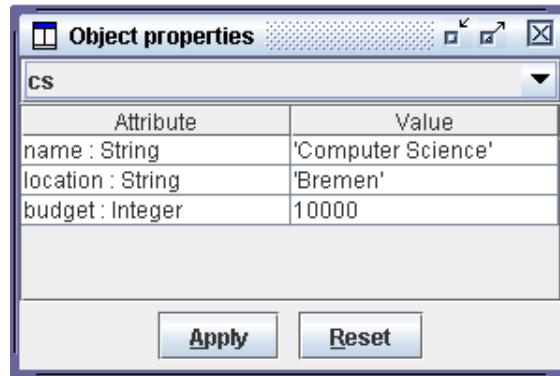
Figure 3.3: Object Properties View

We continue by adding two *Employee* objects and setting their attributes.[1]

```
use> !create john : Employee
use> !set john.name := 'John'
use> !set john.salary := 4000
use> !create frank : Employee
use> !set frank.name := 'Frank'
use> !set frank.salary := 4500
```

Now we have three objects, a department and two employees, but still no connections between them. The layout in the object diagram is continuously refined and updated. This can be turned off by deselecting the option `Auto-Layout` in the context menu of the object diagram.

The previous commands resulted in an invalid system state. This is discussed in detail in the next section.

3.1.1 Model Inherent Constraints

The invariant view indicates some problem with the new system state. The message says: `Model inherent constraints violated`. Model inherent constraints are constraints defined implicitly by a UML model (in contrast to explicit OCL constraints). The details about this message are shown in the log panel at the bottom of the screen. (see figure 4.5) They are also available by issuing a check command at the prompt:

```
use> check
checking structure...
Multiplicity constraint violation in association 'WorksIn':
  Object 'frank' of class 'Employee' is connected to 0 objects of
    class 'Department' via role 'department'
```

---

[1]Again, we use the command line interface here, but the same can be achieved by using the previously discussed steps in the graphical user interface.

```
  but the multiplicity is specified as '1..*'.
Multiplicity constraint violation in association 'WorksIn':
  Object 'john' of class 'Employee' is connected to 0 objects of
    class 'Department' via role 'department'
  but the multiplicity is specified as '1..*'.
...
```

The problem here is that we have specified in the model that each employee has to be related to at least one department object. (see the class diagram shown in figure 1.9) In our current state, no employee has a link to a department. In order to fix this, we insert the missing links into the *WorksIn* association:

```
use> !insert (john,cs) into WorksIn
use> !insert (frank,cs) into WorksIn
```

Links can also be inserted by selecting the objects to be connected in the object diagram and choosing the `insert` command from the context menu.

The new state shows the links in the object diagram as red edges between the *Employee* objects and the *Department* object. (see figure 3.4)

3.2 Validating Invariants

We have seen that class invariants are checked automatically each time a system state changes. This section shows how invariants can be analyzed. We continue the example by adding two projects and linking them to the existing employees and the department.

```
use> !create research : Project
use> !set research.name := 'Research'
use> !set research.budget := 12000
use>
use> !create teaching : Project
use> !set teaching.name := 'Validating UML'
use> !set teaching.budget := 3000
use>
use> !insert (cs,research) into Controls
use> !insert (cs,teaching) into Controls
use>
use> !insert (frank,research) into WorksOn
use> !insert (frank,teaching) into WorksOn
use> !insert (john,research) into WorksOn
```

The resulting state is shown in figure 3.5. In this state, three of the four invariants are true but one fails. The failing one has the name *BudgetWithinDepartmentBudget*. This invariant states that the budget of a project must not exceed the budget of the controlling department. Obviously, one
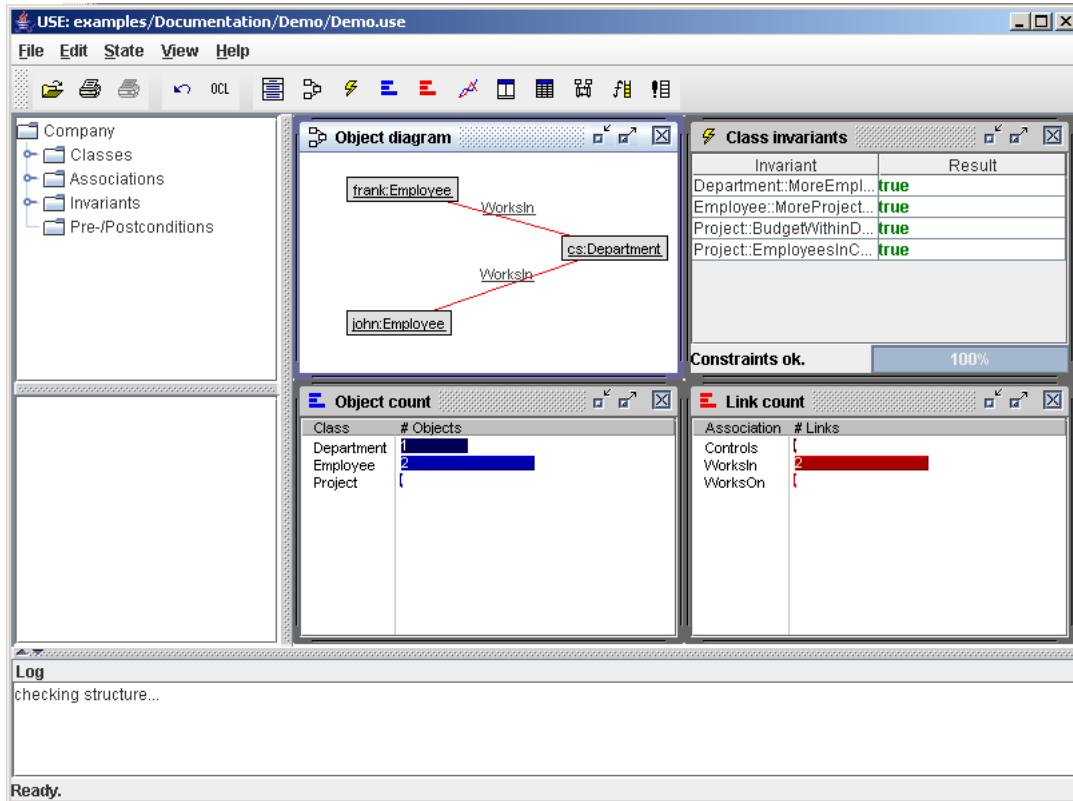
Figure 3.4: Main window after creating the objects and inserting the links

of the two projects in our example must have a budget higher than the budget of the department. The value *false* finally resulting from an evaluation of an invariant is not very helpful in finding the reason for an illegal system state. An Evaluation Browser provides a more detailed view of an expression by showing the results of all sub expressions. (see section 4.5) Double clicking on an invariant will open an Evaluation Browser (see figure 3.6).

The root node in the evaluation browser shows the complete expression and its result, which is false for the chosen invariant. For each component of an expression there are child nodes displaying the sub expressions and their results. You can see that the argument expression of the *forAll* quantifier is false, thus making the whole expression result false. In this sub expression, the variable *self* is bound to the object *research*. The Evaluation Browser has helped to find out that it is the *budget* attribute value of this object which causes the invariant to fail.

3.3 Validating Pre- and Postconditions

OCL provides special syntax for specifying pre- and postconditions on operations in a UML model. Pre- and postconditions are constraints that define a contract that an implementation of the operation has to fulfill. A precondition must hold when an operation is called, a postcondition

34
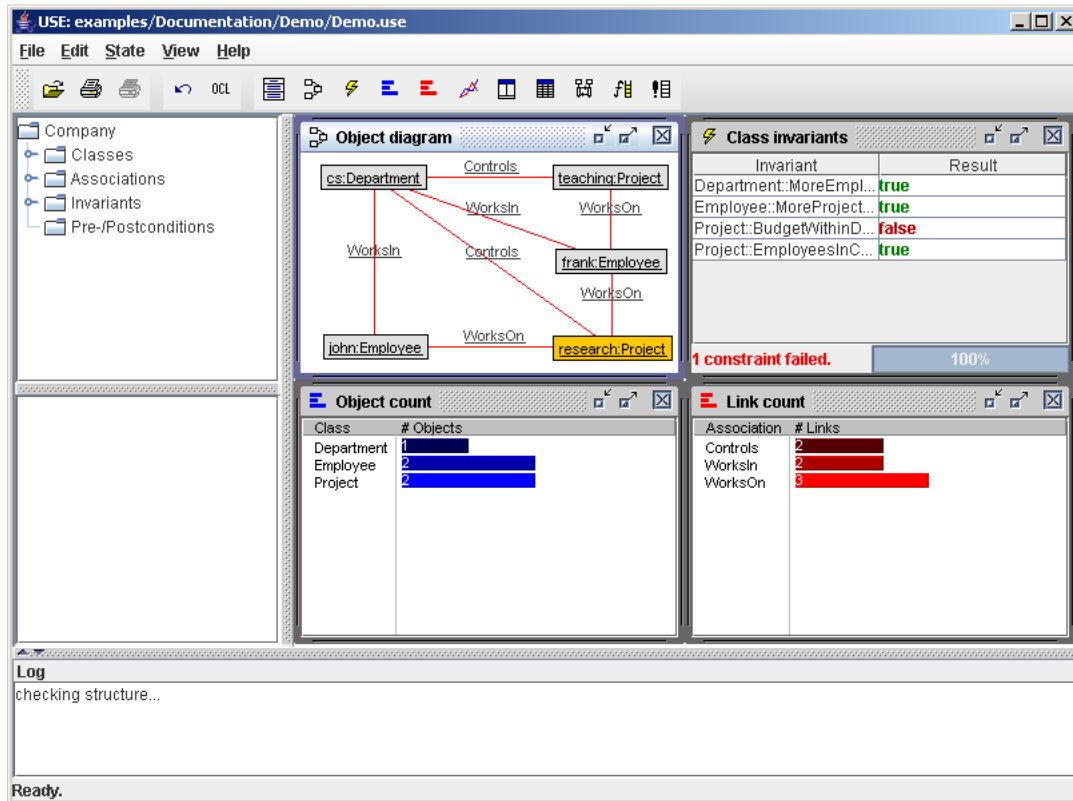
Figure 3.5: Main window after reading in the whole Demo.cmd file

must be true when the operation returns. The USE tool allows to validate pre- and postconditions by simulating operation calls. The following describes how this feature works.

3.3.1 Validating the Person & Company Model

We test the pre- and postconditions of the Person & Company example specified in section 2.3.2. First we start the USE tool with the specification of the example model.

```
use> open ../examples/Documentation/Employee/Employee.use
compiling specification...
Model Employee (2 classes, 1 association, 0 invariants, 3 operations,
  7 pre-/postconditions)
```

At the prompt, we enter the following commands to create two objects.

```
use> !create ibm : Company
use> !create joe : Person
use> !set joe.name := 'Joe'
use> !set joe.age := 23
```
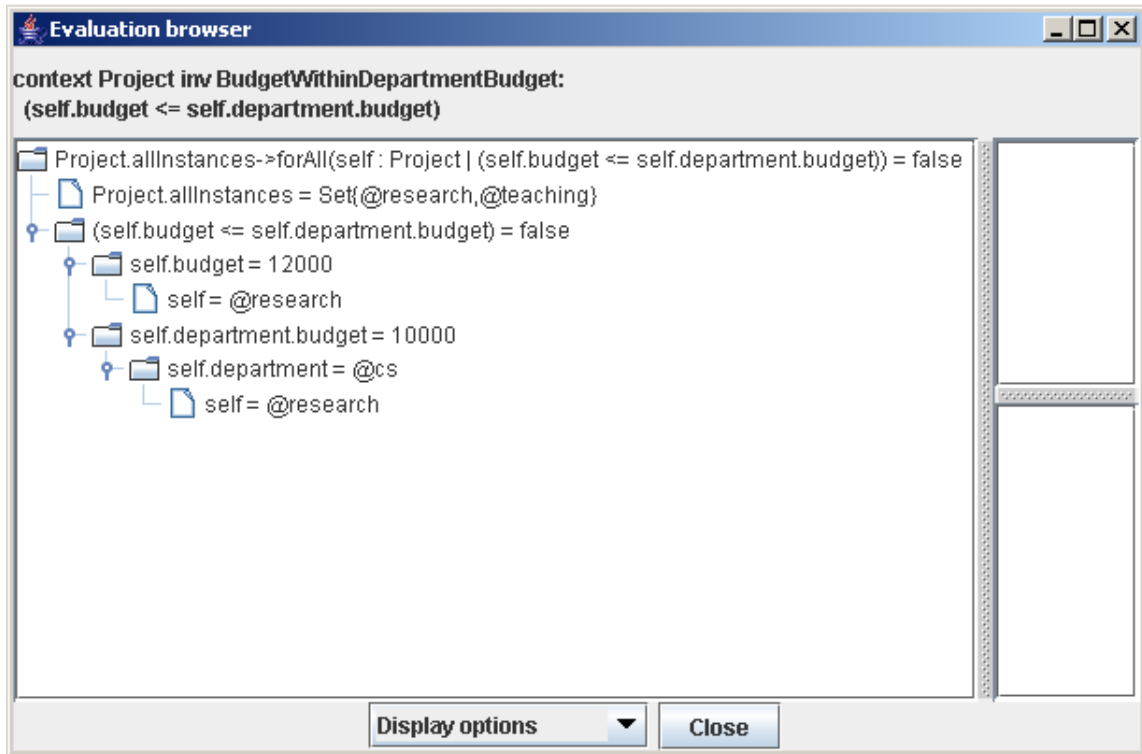
Figure 3.6: Evaluation Browser showing the violated constraint

The current system state can be visualized with an object diagram view in USE (see figure 3.7).

Next, we want to call the operation *hire* on the company object to hire *joe* as a new employee.

Calling Operations and Checking Preconditions

Operation calls are initiated with the command `openter`. Its syntax is presented in section 5.1.12. The following command shows the top level bindings of variables. These variables can be used in expressions to refer to existing objects.

```
use> info vars
ibm : Company = @ibm
joe : Person = @joe
```

We invoke the operation *hire* on the receiver object *ibm* and pass the object *joe* as parameter.

```
use> !openter ibm hire(joe)
precondition 'hirePre1' is true
precondition 'hirePre2' is true
```

The openter command has the following effect.
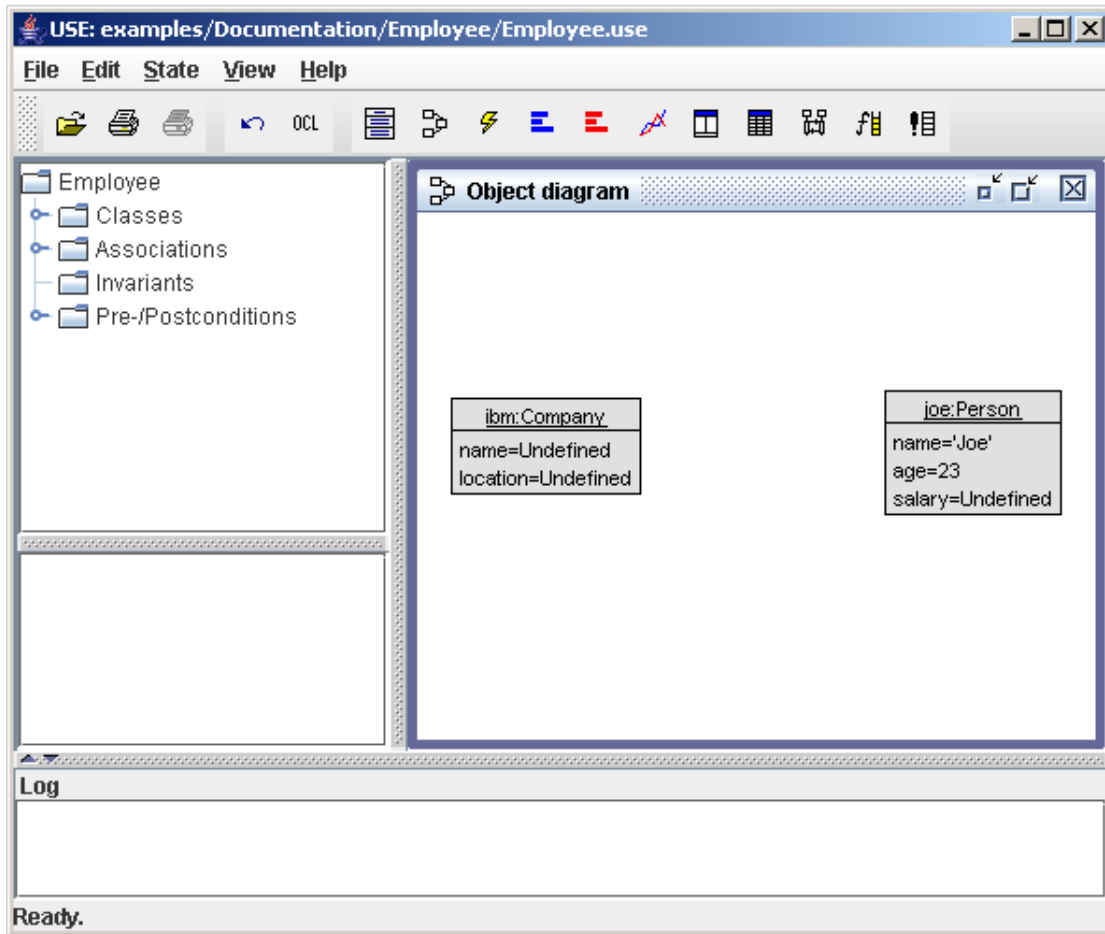
Figure 3.7: Object diagram of the Person & Company example

1. The source expression is evaluated to determine the receiver object.

2. The argument expressions are evaluated.

3. The variable *self* is bound to the receiver object and the argument values are bound to the formal parameters of the operation. These bindings determine the local scope of the operation.

4. All preconditions specified for the operation are evaluated.

5. If all preconditions are satisfied, the current system state is saved and the operation call is saved on a call stack.

In the example, the call of the operation *hire* was successful because both preconditions are satisfied. The stack of currently active operations can be viewed by issuing the following command.

```
use> info opstack
```

```
active operations:
1. Company::hire(p : Person)
```

We can verify the bindings of the *self* variable and the formal parameter *p* as follows.

```
use> info vars
ibm : Company = @ibm
joe : Person = @joe
p : Person = @joe
self : Company = @ibm
```

Operation Effects

We can simulate the execution of an operation with the usual USE primitives for changing a system state. The postcondition of the *hire* operation requires that a *WorksFor* link between the person and the company has to be created. We also set the salary of the new employee.

```
use> !insert (p, ibm) into WorksFor
use> !set p.salary := 2000
```

The object diagram in 3.8 shows the new system state with the link between the *Person* and *Company* objects.

Exiting Operations and Checking Postconditions

After generating all side effects of an operation, we are ready to exit the operation and check its postconditions. The command `opexit` simulates a return from the currently active operation. The syntax is:

`!opexit` *ReturnValExpr*

The optional *ReturnValExpr* is only required for operations with a result value. An example will be given later. The operation *hire* specifies no result, so we can just issue:

```
use> !opexit
postcondition 'hirePost' is true
```

The opexit command has the following effect.

1. The currently active operation is popped from the call stack.

2. If an optional result value is given, it is bound to the special OCL variable *result*.

3. All postconditions specified for the operation are evaluated in context of the current system state and the pre state saved at operation entry time.

4. All local variable bindings are removed.
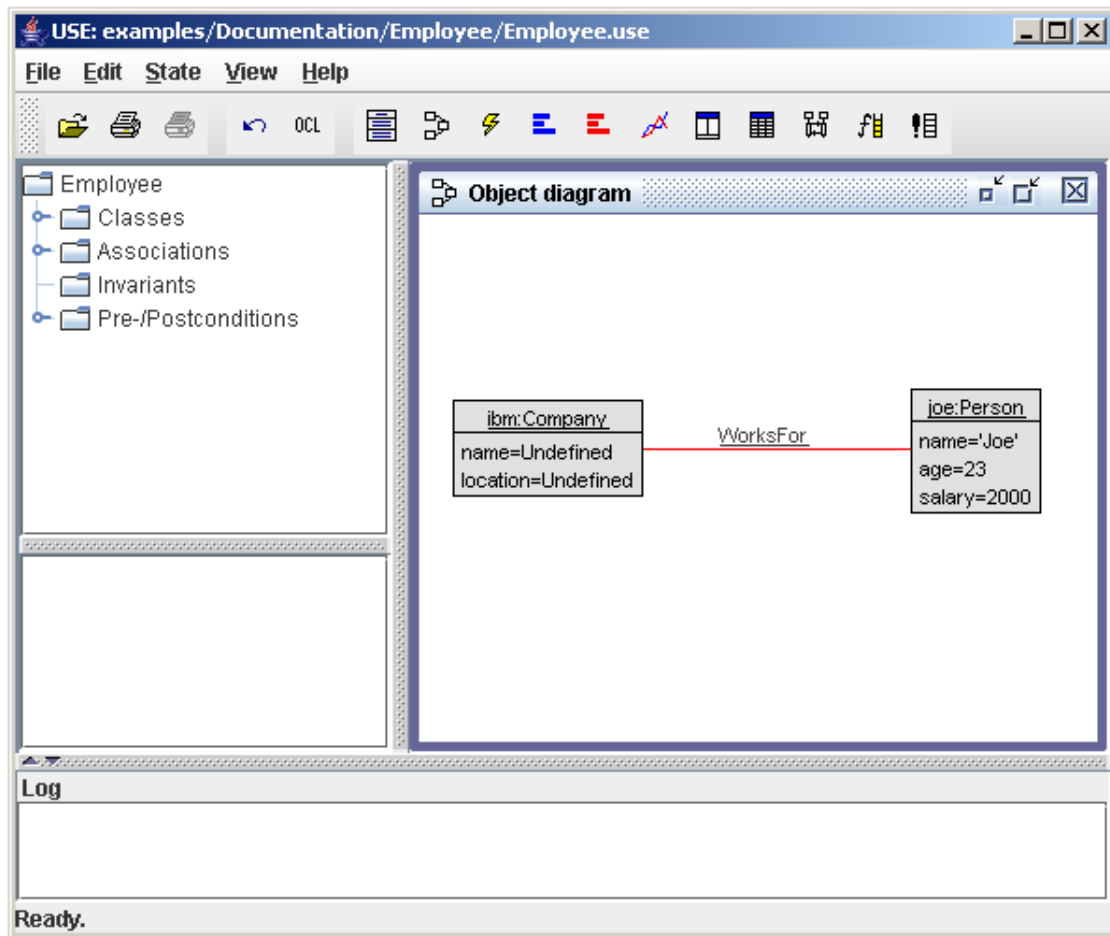
Figure 3.8: Object diagram of the Person & Company example after changing the state

In our example, the postcondition *hirePost* is satisfied.

The operation has been removed from the call stack:

```
use> info opstack
no active operations.
```

All variable bindings local to operations are removed on exit.

```
use> info vars
ibm : Company = @ibm
joe : Person = @joe
```

Note that object names are elements of the top level bindings. If you create new objects inside an operation call, their names will still be available after exiting the operation.

Result Values and References to the Previous State

The operation *raiseSalary* in class *Person* is used for changing the salary of an employee by a given rate. The following constraints are added to the model specification.

```
context Person::raiseSalary(rate : Real) : Real
  post raiseSalaryPost:
    salary = salary@pre * (1.0 + rate)
  post resultPost:
    result = salary
```

The first postcondition *raiseSalaryPost* requires that the new value of the salary attribute equals a value that is computed in terms of the previous value using the *@pre* modifier. The second postcondition *resultPost* specifies that the result value of the operation equals the new salary.

We call *raiseSalary* on the new employee *joe*. The rate 0.1 is given to raise the salary by 10%.

```
use> !openter joe raiseSalary(0.1)
```

The *salary* attribute is assigned a new value with the `set` command.

```
use> !set self.salary := self.salary + self.salary * rate
```

Since *raiseSalary* is an operation with a return value, we have to specify a result value on exit. This value is bound to the OCL result variable when the postconditions are evaluated.

```
use> !opexit 2200
postcondition 'raiseSalaryPost' is true
postcondition 'resultPost' is true
```

Visualization as Sequence Diagram

The USE tool can visualize a sequence of operation calls as a UML sequence diagram. The screenshot in figure 3.9 shows the sequence of calls for the example. During a validation session the diagram is automatically updated on each operation call.

3.3.2 An Example with oclIsNew

The graph model specified in section 2.3.3 includes constraints calling the operation *oclIsNew*. We use the following command script for animating the model. The script simulates three operation calls. The first one is expected to succeed while the second and third one should violate the postconditions.

```
!create n1 : Node

-- this call satisfies both postconditions
!openter n1 newTarget()
!create n2 : Node
```
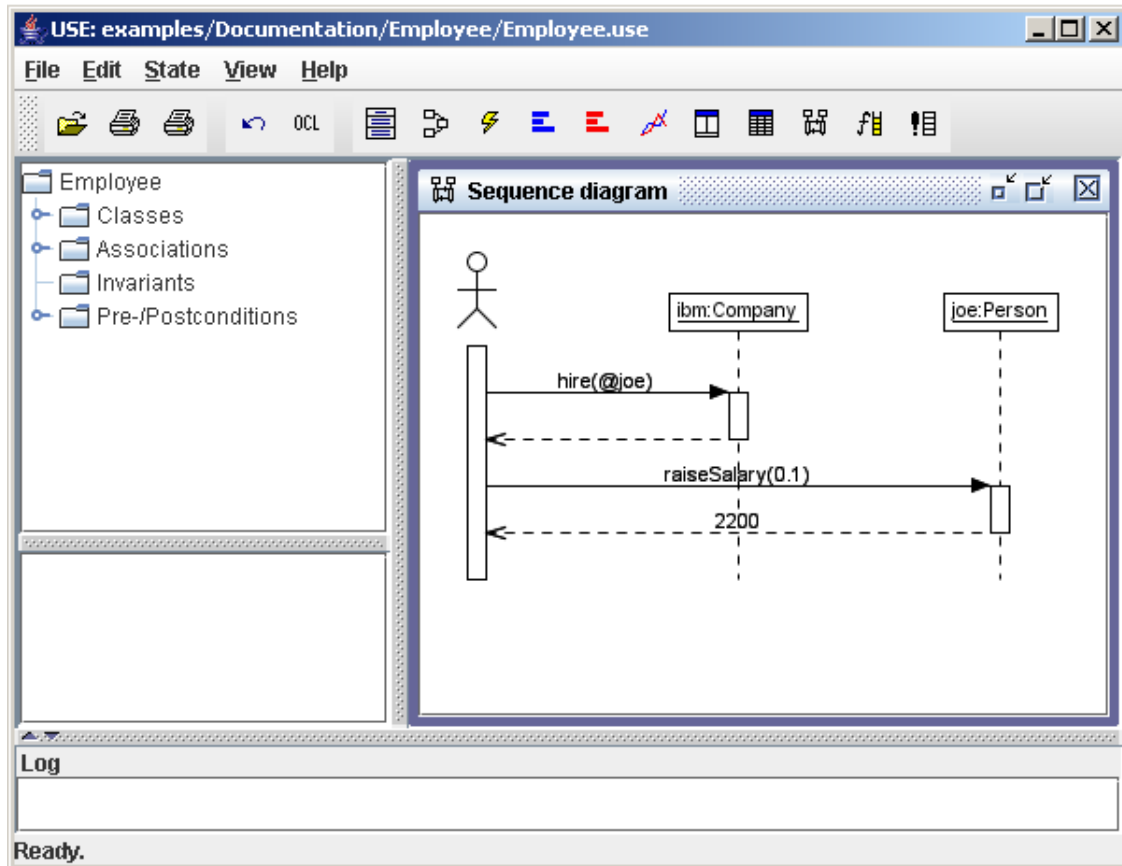
Figure 3.9: Sequence diagram of the Person & Company example

```
!insert (n1,n2) into Edge
!opexit

-- postcondition oneNewTarget fails,
-- because n3 is not linked to n1
!openter n1 newTarget()
!create n3 : Node
!opexit

-- postcondition targetNodeIsNew fails,
-- because n3 has already been created above
!openter n1 newTarget()
!insert (n1,n3) into Edge
!opexit
```

Here is the output of the USE tool confirming our expectations about the success and failure of postconditions. Details during the evaluation of failing postconditions provide hints about what

went wrong.

```
$ use -nogui ../examples/Documentation/Graph/Graph.use
  ../examples/Documentation/Graph/Graph.cmd

use version 2.3.1-devel, Copyright (C) 1999-2006 University of Bremen
compiling specification...
Model Graph (1 class, 1 association, 0 invariants, 1 operation, 2 pre-/postconditions)
Graph.cmd> -- Opens the class diagram:
Graph.cmd> -- open examples/Documentation/Graph/Graph.use
Graph.cmd>
Graph.cmd> -- Creates the object diagram:
Graph.cmd> -- read examples/Documentation/Graph/Graph.cmd
Graph.cmd>
Graph.cmd> !create n1 : Node
Graph.cmd>
Graph.cmd> -- this call satisfies both postconditions
Graph.cmd> !openter n1 newTarget()
Graph.cmd> !create n2 : Node
Graph.cmd> !insert (n1,n2) into Edge
Graph.cmd> !opexit
postcondition 'oneNewTarget' is true
postcondition 'targetNodeIsNew' is true
Graph.cmd>
Graph.cmd> -- postcondition oneNewTarget fails,
because n3 is not linked to n1
Graph.cmd> !openter n1 newTarget()
Graph.cmd> !create n3 : Node
Graph.cmd> !opexit
postcondition 'oneNewTarget' is false
evaluation results:
  self : Node = @n1
  self.target : Set(Node) = Set{@n2}
  self : Node = @n1
  self.target@pre : Set(Node) = Set{@n2}
  (self.target - self.target@pre) : Set(Node) = Set{}
  (self.target - self.target@pre)->size : Integer = 0
  1 : Integer = 1
  ((self.target - self.target@pre)->size = 1) : Boolean = false
postcondition 'targetNodeIsNew' is true
Graph.cmd>
Graph.cmd> -- postcondition targetNodeIsNew fails,
because n3 has already been create above
Graph.cmd> !openter n1 newTarget()
```

```
Graph.cmd> !insert (n1,n3) into Edge
Graph.cmd> !opexit
postcondition 'oneNewTarget' is true
postcondition 'targetNodeIsNew' is false
evaluation results:
  self : Node = @n1
  self.target : Set(Node) = Set{@n2,@n3}
  self : Node = @n1
  self.target@pre : Set(Node) = Set{@n2}
  (self.target - self.target@pre) : Set(Node) = Set{@n3}
  n : Node = @n3
  n.oclIsNew : Boolean = false
  (self.target - self.target@pre)->forAll(n : Node | n.oclIsNew) : Boolean = false
Graph.cmd>
use>
```

The screenshot in figure 3.10 shows the sequence diagram automatically generated from the example. Dashed return arrows in red indicate that a postcondition failed on exit from an operation call.

3.3.3 Nested Operation Calls

The factorial example specified in section 2.3.4 includes nested operation calls. The following commands show the computation of 3!.

```
use> !create r : Rec
use> !openter r fac(3)
precondition 'pre1' is true
use> !openter r fac(2)
precondition 'pre1' is true
use> !openter r fac(1)
precondition 'pre1' is true
```

The call stack now looks like this.

```
use> info opstack
active operations:
1. Rec::fac(n : Integer) : Integer
2. Rec::fac(n : Integer) : Integer
3. Rec::fac(n : Integer) : Integer
```

We exit the operations in reverse order and provide result values that satisfy the postcondition.

```
use> !opexit 1
postcondition 'post1' is true
use> !opexit 2
```

43
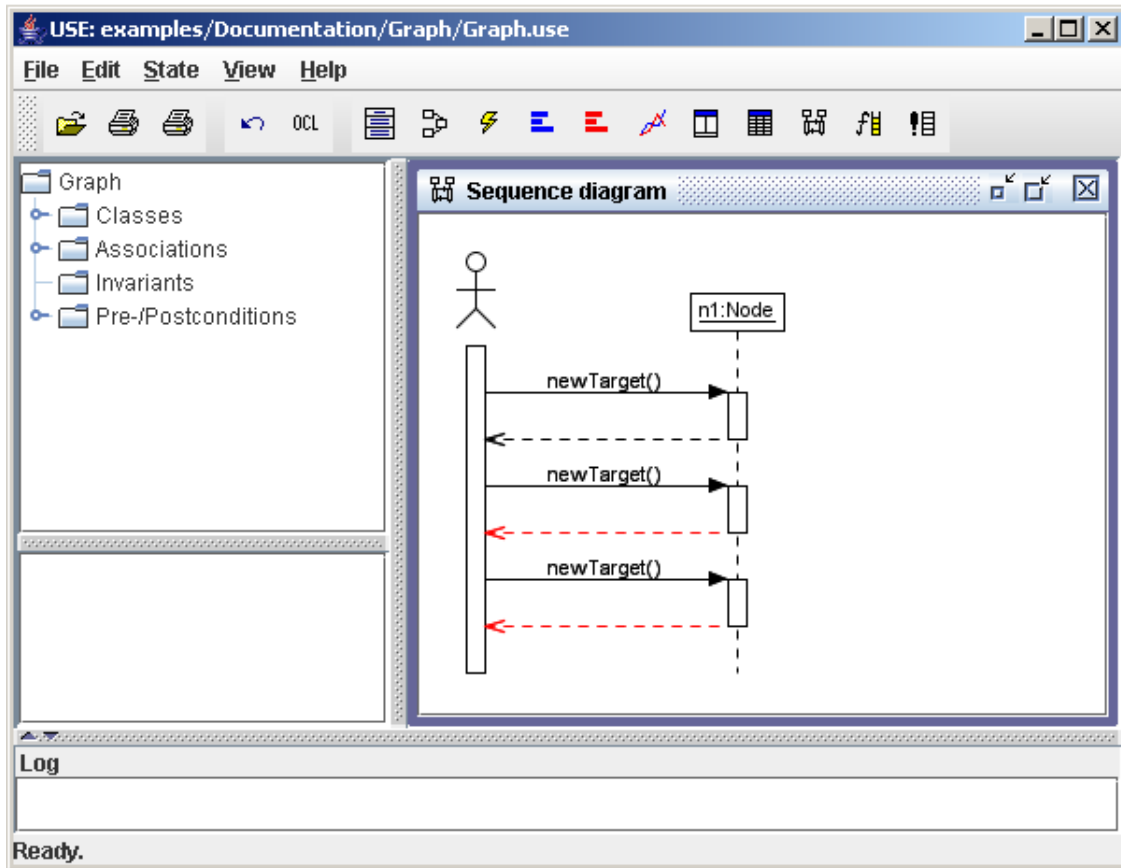
Figure 3.10: Sequence diagram of the Graph example

```
postcondition 'post1' is true
use> !opexit 6
postcondition 'post1' is true
```

The screenshot in figure 3.11 shows the sequence diagram automatically generated from the example. Note the stacked activation frames resulting from the recursion.
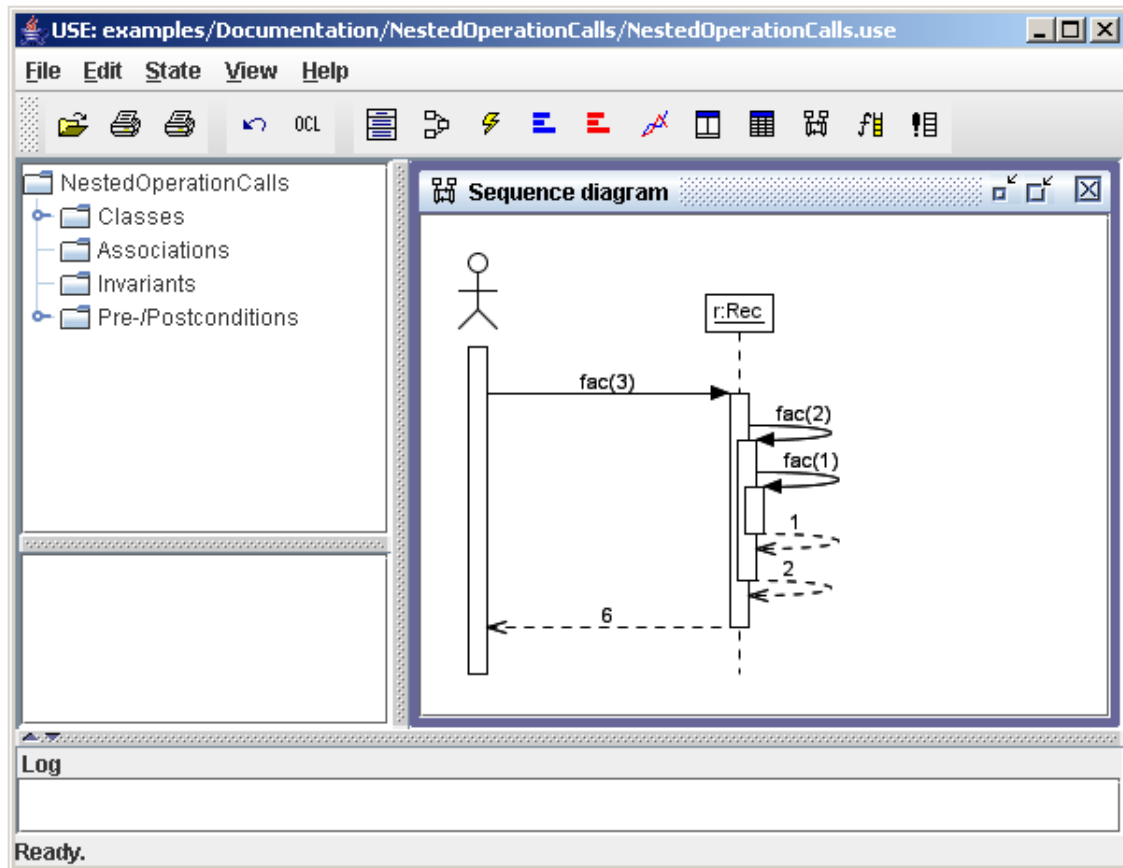
Figure 3.11: Sequence diagram of the factorial example

4 GUI Reference

4.1 The Menubar

The symbols on the left side of the menu entries indicate that there is a corresponding button available at the toolbar.

4.1.1 File

Open Specification...

Loads an available USE specification from file (*filename*`.use`).

Save Script...

Saves all previously entered operation calls and commands which changed the system state to file (*filename*`.cmd`).

Save Protocol...

Saves a detailed protocol including many GUI and shell activities.

Printer Setup...

Opens a dialog with modifiable standard printer settings.

Print Diagram...

Opens the printer window for printing the active diagram with any desired settings.

Print View...

This function is enabled for sequence diagrams only. It prints the visible part of the diagram. For printing the whole sequence diagram, the `Print Diagram...` function has to be used.

Exit

Quits the running USE system.

4.1.2 Edit

Undo

This function cancels the commands changing the system state step by step. It makes no difference between GUI and shell commands.

4.1.3 State

Create object...

Shows all specified classes. After selecting a class, any number of objects may be created by entering an unique object name.

Evaluate OCL expression...

This function opens an evaluating window which consists of two parts. In the upper part you can enter a OCL expression. After evaluating the expression the lower part shows the result with its type. The `Clear` button clears the result information. The `Browser` button opens the evaluation browser for analyzing the entered expression and its parts. If the evaluation browser is still opened and a new expression is evaluated the browser will be actualized. If the expression cannot be evaluated the browser window will be closed. This also happens if the `Clear` button is pressed.
The model elements like class names, role names etc. may be integrated into the OCL expression. If a system state is defined, object names may be used too.

Check structure now

This function checks if all multiplicities defined in the specification are fulfilled by the system state.

Check structure after every change

Checks the structure for every command, which changed the system state.

Reset

Resets the system state to the empty state.

4.1.4 View

Create

Creates one of the available diagram views.

Tile

Arranges all displayed views next to each other.

Close all

Closes all displayed diagram views.

4.1.5 Help

About...

Opens the About window.

4.2 Toolbar

Open Specification (see section 4.1.1)

Print Diagram (see section 4.1.1)

Print View (see section 4.1.1)

Undo (see section 4.1.2)

OCL Evaluate OCL expression (see section 4.1.3)

Create Class Diagram View (see section 4.4.2)

Create Object Diagram View (see section 4.4.3)

Create Class Invariant View (see section 4.4.4)

Create Object Count View (see section 4.4.5)

Create Link Count View (see section 4.4.6)

Create State Evolution View (see section 4.4.7)

Create Object Properties View (see section 4.4.8)

Create Class Extend View (see section 4.4.9)

Create Sequence Diagram View (see section 4.4.10)

Create Call Stack View (see section 4.4.11)

Create Command List View (see section 4.4.12)
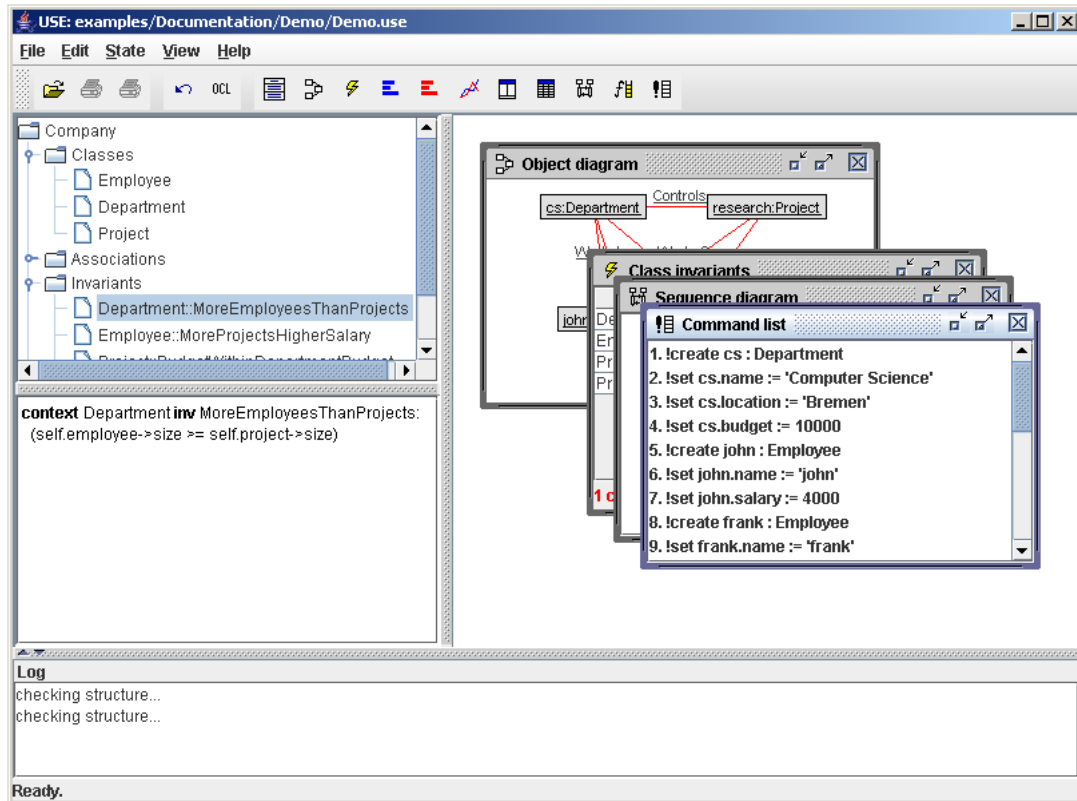
Figure 4.1: Main window

4.3 The Main Window

4.3.1 Showing the diagram views

The main part of the window shows the opened diagram views.

4.3.2 Overview of the Specification

The top left window represents the model browser. (see figure 4.2) It shows all defined classes, associations, invariants and pre- and postconditions. A right click into this part of the main window opens a context menu. (see figure 4.3) The menu provides the possibility to sort the specification elements e.g. by name or use file order.

4.3.3 Definition of the Specification elements

The window below the specification overview shows the definition of the selected specification element as it is defined in the .use file. (see figure 4.4)

49
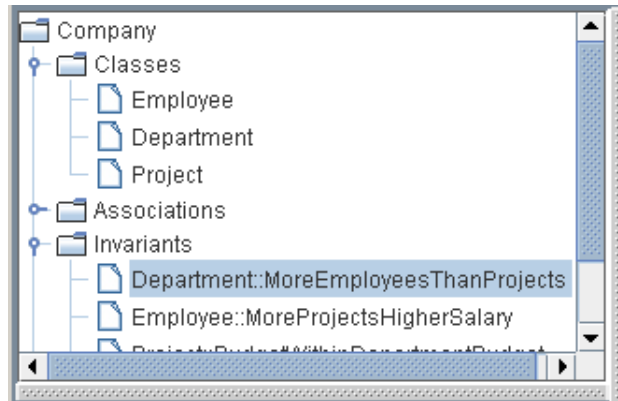
Figure 4.2: Overview of the Specification
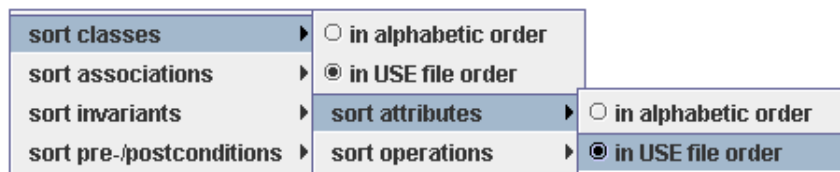
Figure 4.3: Sorting

4.3.4 Log window

The lower part of the main window show a log of the system activities. (see figure 4.5) It also lists possible syntax resp. type check errors found in the loaded specification and structural errors. Click right to clear the log window.

4.3.5 Status and Tips

A line on the bottom of the main window shows the current USE status and tips with reference to the displayed diagrams. (see figure 4.6)

4.4 Diagram Views

There are different views, which can be used to analyze the current system state with reference to the specification.

4.4.1 General Functions

Double click on the head of an active view to maximize the window resp. reduce it to the previous size. There are three symbols in the upper right corner. They minimize, maximize resp. reduce or close the active window.

50
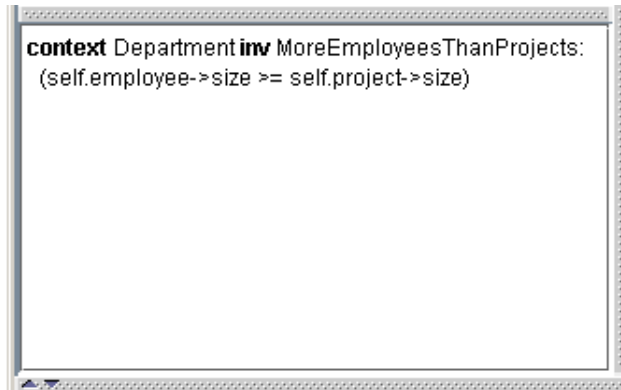
Figure 4.4: Definition of the specification elements
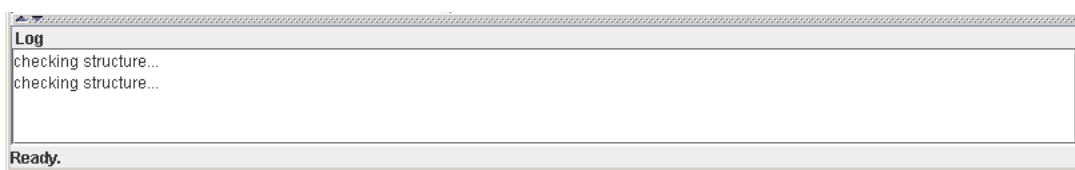
Figure 4.5: Log window

4.4.2 Class Diagram View

This view shows the class diagram defined by the loaded specification. (see figure 4.7) It displays classes, attributes, operations, associations, inheritance, compositions, aggregations, association classes, enumerations, role names and multiplicities. Associations connect classes via edges. It is possible to create movable nodes by double clicking an edge. If an association is not 2 ary a diamond will connect the three or more participating classes. A dashed line connects association classes to their edge. You can move elements like classes, diamonds, role names, multiplicities etc.. If you select an element, it appears orange. To mark more than one element hold the shift button and select the elements. The selected elements can be moved together.

Figure 4.7 shows the context menu, which is displayed after a right click. You can choose if associations, role names, multiplicities, attributes or operations should be displayed. If you enable the `Auto-Layout` option, the system tries to arrange the class diagram elements optimally. The `Anti-aliasing` option switches the anti aliasing on and off. It is possible to save resp. load an diagram layout to resp. from file (*filename*`.clt`).

If you select at least one Element, you can hide it or all other elements but the selected ones. The `Show` command recovers the hidden elements. (see figure 4.8)

4.4.3 Object Diagram View

The Object Diagram View shows the object diagram defined by the actual system state. (see figure 4.9) It shows objects, attributes, attribute values, links, association names and role names. The general functions are similar to the Class Diagram View functions presented in section 4.4.2.

Figure 4.6: Status and tips
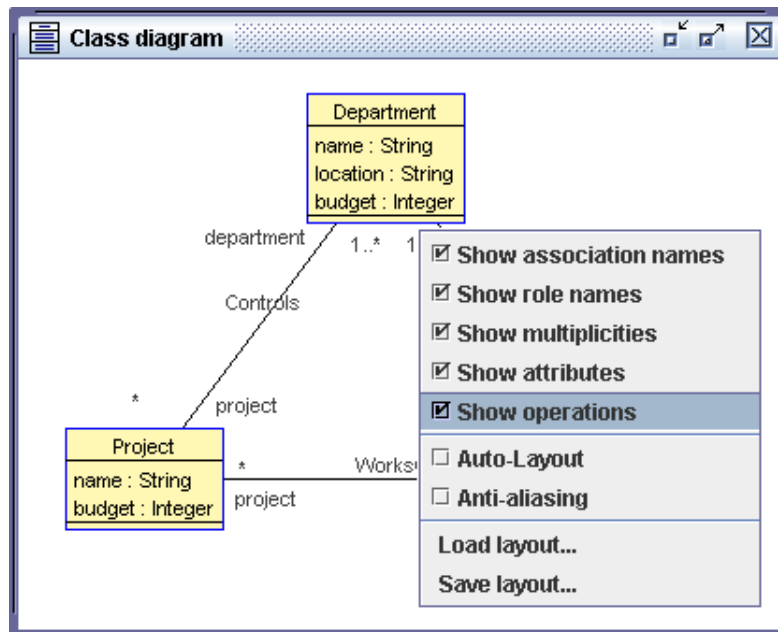
Figure 4.7: Class Diagram View (Employees, Departments and Projects Example)

Objects are displayed with object name and its type. Double click on an object to open the object properties view. (see figure 4.4.8) Links are represented by a red line. Association names, role names and attributes are optional elements. To display them, check the corresponding box in the context menu, which is shown after a right click. The `Auto-Layout`, `Anti-aliasing`, `Load layout...` and `Save layout...` functions are explained in section 4.4.2.

Creating and destroying objects without Shell commands

The Model Browser (see section 4.3.3) shows all specified classes. To create an instance of a class just drag the class name and drop it into the object diagram view. This creates an object with undefined attribute values.

You can destroy existing objects by selecting them. The context menu shows a new `Delete` function, which will destroy the object and the participating links. (see figure 4.10)

Inserting and deleting links without shell commands

To insert a link between two or more objects, select the objects and open the context menu. Hold the shift button to select more than one object. If there is an appropriate association the context menu shows an `insert` command, which inserts a link between the selected objects. Associa-
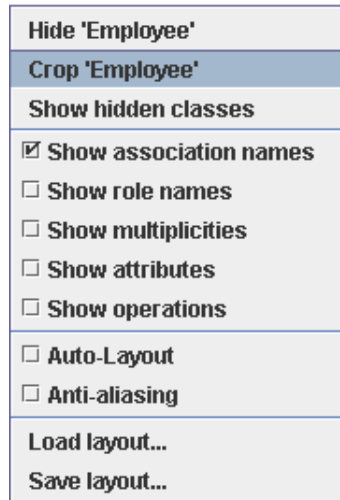
Figure 4.8: Class Diagram View - Context Menu with Hide, Crop, and Show

tion classes are created the same way.

Remove a link by selecting the involved objects. The context menu shows a `delete` function.

4.4.4 Class Invariant View

Shows all invariants. (see figure 4.11) If no instance of the invariant context violates the corresponding invariant and no model inherent constraint (see section 3.1.1) the view shows *true*. If an objects violates a model inherent constraint it appears *N/A*. *false* appears otherwise. The bottom of the window shows the number of violated invariants in the actual system state. A double click opens the evaluation browser analyzing the current invariant with respect to the actual system state. (see section 4.5)

4.4.5 Object Count View

Shows all classes on the left side and the number of their instances on the right side. A bar chart shows an overview of the number of instances.

4.4.6 Link Count View

Shows all associations on the left side and the number of links on the right side. A bar chart shows an overview of the number of links.

4.4.7 State Evolution View

Shows a line chart. (see figure 4.14) A blue line represents objects and a red line represents links in the actual system state. The y axis represents the number of objects resp. links. The
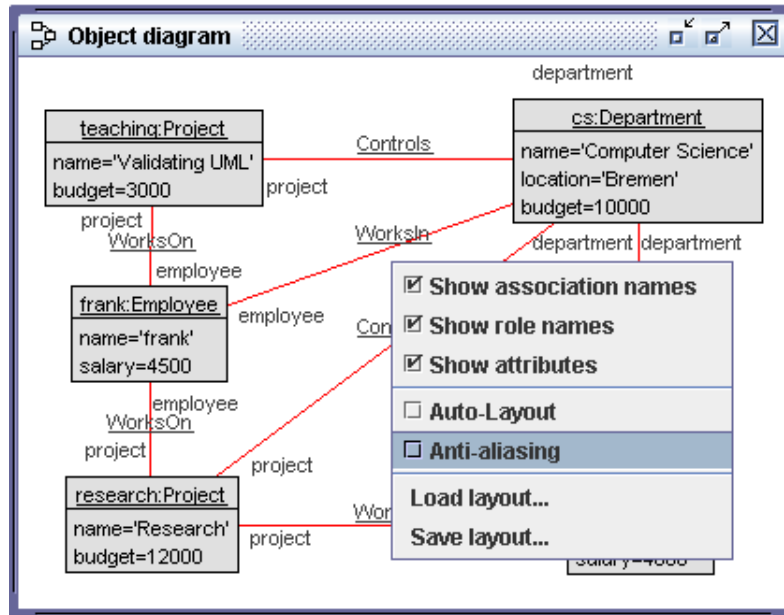
Figure 4.9: Object Diagram View (Employees, Departments and Projects Example)

x axis represents the number of changes the user made. Commands like `set` are considered as changes, too.

4.4.8 Object Properties View

The drop down menu of this view includes all objects. (see figure 4.15) If you choose an object its attributes and their values are displayed. Double click on a value to change it. The `Apply` button saves the changes.

4.4.9 Class Extend View

This view shows all objects of the selected class and their attribute values. (see figure 4.16) A right click opens a context menu. You can switch on the invariant results for each object and select a class. An invariant receives a check symbol if the given object does not violate it. Otherwise a cross appears. If an object violates model inherent constraints the invariant is not evaluated for this object. Then a question mark appears. A double click opens the evaluation browser with the evaluation of the selected invariant. It marks the sub formula for the corresponding object.

4.4.10 Sequence Diagram View

Description will be available in the next document version.
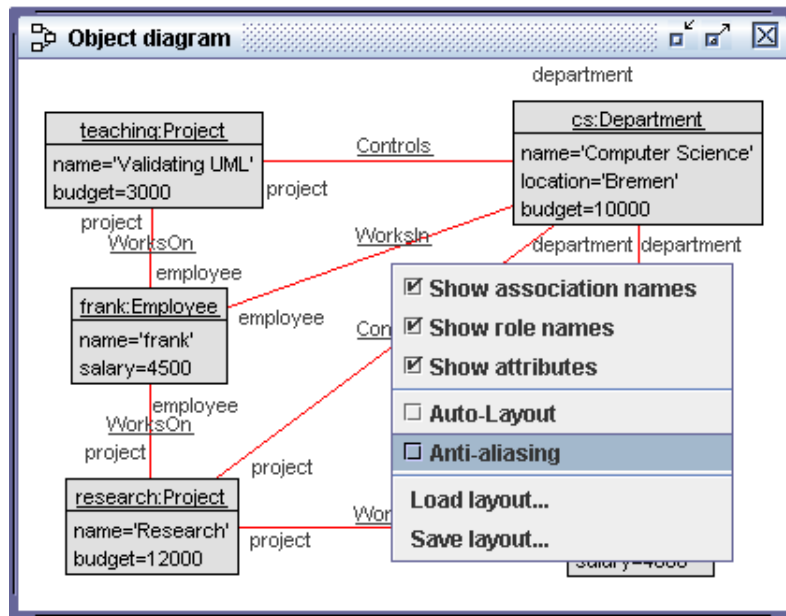
Figure 4.10: Object Diagram View (Employees, Departments and Projects Example)

Figure 4.11: Class Invariant View (Employees, Departments and Projects Example)

Figure 4.12: Object Count View (Employees, Departments and Projects Example)
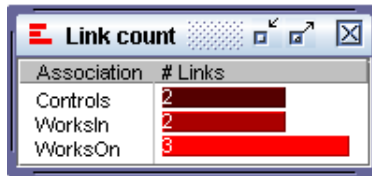
Figure 4.13: Link Count View (Employees, Departments and Projects Example)
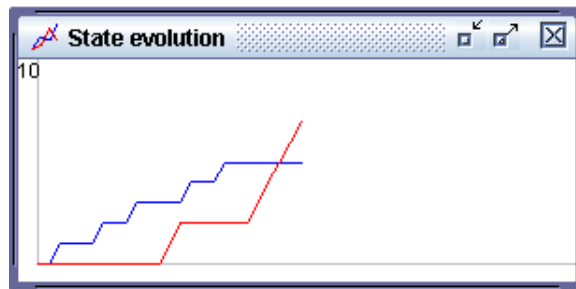
Figure 4.14: State Evolution View (Employees, Departments and Projects Example)
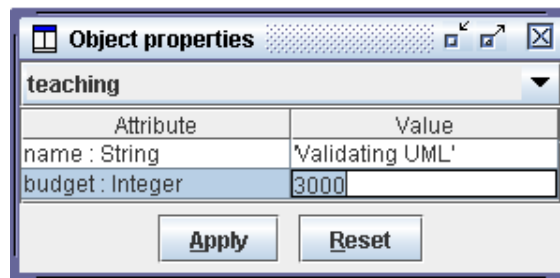
Figure 4.15: Object Properties View (Employees, Departments and Projects Example)
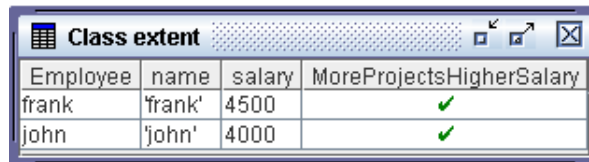
Figure 4.16: Class Extend View (Employees, Departments and Projects Example)
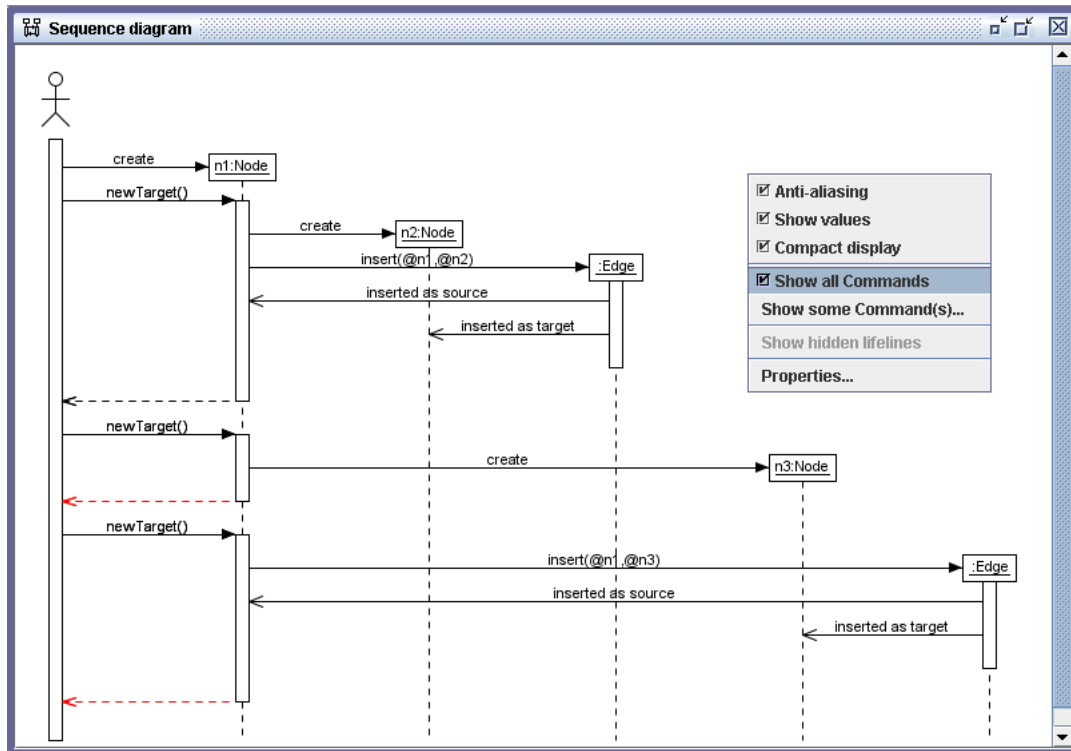
Figure 4.17: Sequence Diagram View (Graph Example)

4.4.11 Call Stack View

The Call Stack lists all operations which were called with the `openter` command and did not terminate yet. They terminate if you use the `opexit` command. A right click opens the context menu. You can choose if the operation signature or the concrete operation call should be displayed.

4.4.12 Command List View

This view lists all commands defining the actual system state. (see figure 4.23) The `reset` command resets the system state and empties the command list.

4.5 Evaluation Browser

You can open the Evaluation Browser via the Class Invariant View (see section 4.4.4), via the Class Extend View (see section 4.4.9) or the OCL Evaluation Window (see section 4.1.3). The figure 4.24 shows the Evaluation Browser displaying the evaluation tree for the invariant *MoreEmployeesThanProjects* in the Employees, Departments and Projects example.

Figure 4.18: Choose Commands (Graph Example)
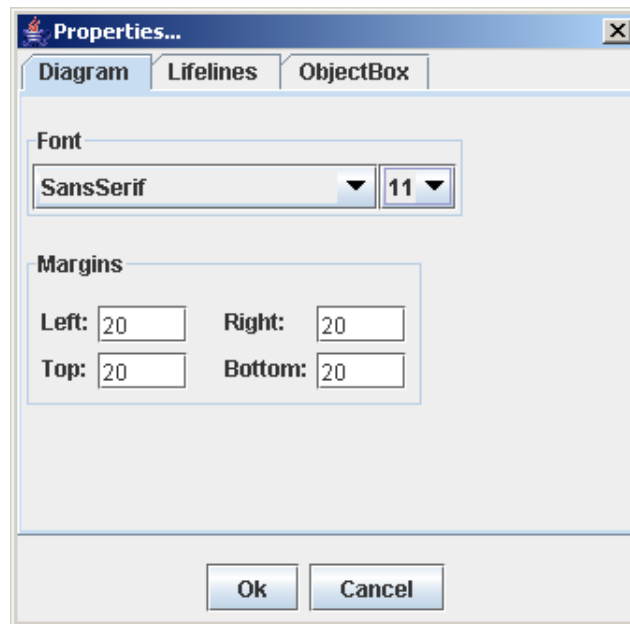
Figure 4.19: Properties - Diagram (Graph Example)

A right click opens a large context menu. Its elements are explained in the following subsections.

4.5.1 Extended Evaluation

You can select which OCL operations (*exists*, *forAll*, *and*, *or*, *implies*) should be evaluated extendedly. (see figure 4.25

exists

Selecting the menu entry `exists` implicates that all *exists* expressions are evaluated extendedly. The extended evaluation does not stop if an element fulfilling the body of the *exists* expression was already found. The whole collection expression is evaluated and all elements fulfilling the expression are displayed.
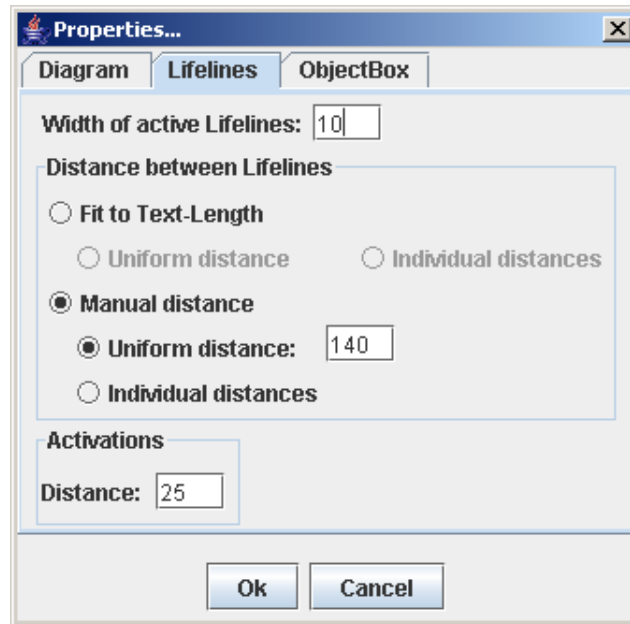
Figure 4.20: Properties - Lifelines (Graph Example)

forAll

If you select the option `forAll` every *forAll* expression is evaluated extendedly. If an element does not fulfill the body of a *forAll* expression, the extended evaluation does not stop, but continues the iteration until the last element was regarded. All elements and their evaluation results are displayed.

and

The extended evaluation of *and* expressions implies, that the right side of an *and* expression is evaluated even if the left side is not true. The result of the right side is always displayed.

or

If the left side of an *or* expression is true, USE normally stops the evaluation of this expression. You can continue the evaluation even though the left side is already true, by selecting the `or` option.

implies

The extended evaluation of *implies* expressions evaluate the conclusions even if the premises are false.
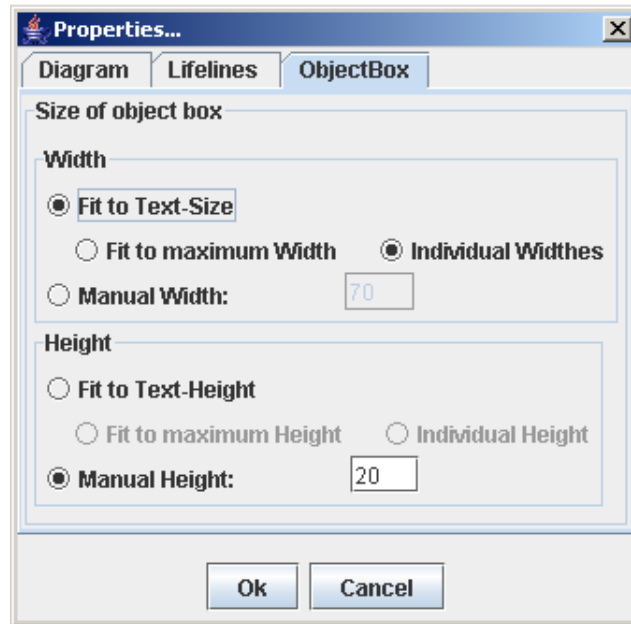
Figure 4.21: Properties - Object Box (Graph Example)

all

The all entry selects all options listed above.

4.5.2 Variable Assignment Window

If you switch on the Variable Assignment Window, it is displayed on the right side of the evaluation Tree. [1] It shows all value assignments of the existing variables in the selected tree node. The example in figure 4.24 shows the variable *self* of type *Department* and its value *@cs*. The corresponding node represents the expression *@cs.project → size = 2*.

4.5.3 Subexpression Evaluation Window

The Subexpression Evaluation Window is displayed on the right side of the evaluation tree resp. below the Variable Assignment Window. It shows the subexpressions of the marked tree node, which are evaluated in the next step. The example in figure 4.24 marks a tree node with expression *@cs.project → size = 2*. The navigation expression has to be evaluated next. So the window shows the result of this subexpression: *Set{@research, @teaching} → size = 2*

4.5.4 Tree Views

You can choose between different tree views. They are listed in figure 4.26 and explained below.

---

[1]If the Subexpression Evaluation Window is displayed too, the Variable Assignment Window appears above it.
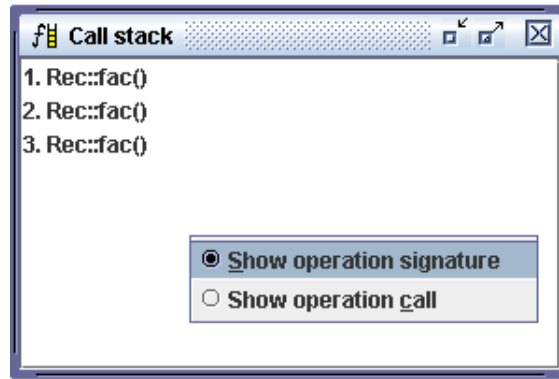
Figure 4.22: Call Stack View (Factorial Example)
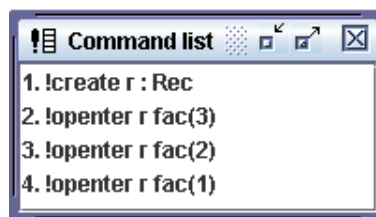
Figure 4.23: Command List View (Factorial Example)

Late Variable Assignment

In this view the tree nodes, showing the variable assignments, are the leafs of the tree. (see figure 4.27)

Early Variable Assignment

This view shows the variable assignment as soon as possible in the evaluation tree. [1] (see figure 4.28) Simultaneous assignments are shown in the same node. They are separated by commas.

Variable Assignment & Substitution

The Variable Assignment & Substitution view is a refinement of the previous presented view. The variable names are substituted by their values. (see figure 4.29)

Variable Substitution

This view is similar to the Variable Assignment & Substitution view, but nodes with variable assignments are not displayed here. (see figure 4.30)

---

[1]Assignments can not be displayed until the variables are bounded in the corresponding OCL expression.
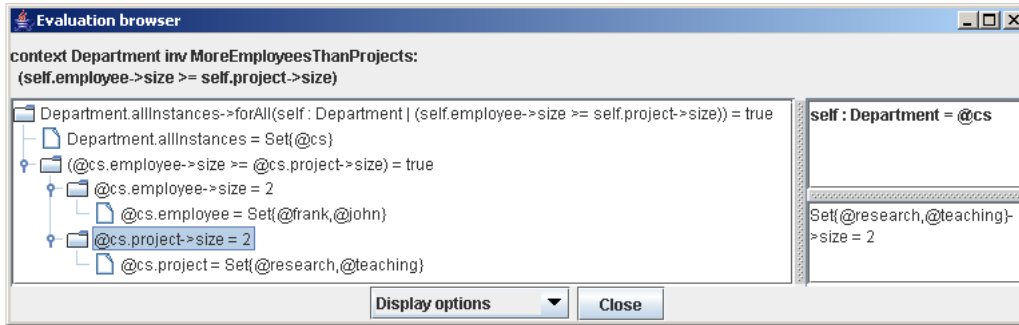
Figure 4.24: Evaluation Browser (Employees, Departments and Projects Example)
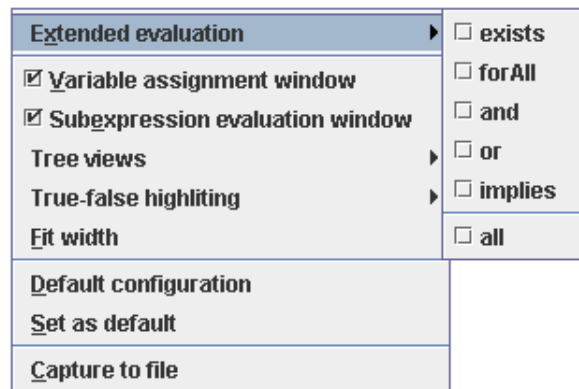
Figure 4.25: Menu - Extended Evaluation

No Variable Assignment

This view does not display tree nodes with variable assignments, and does not substitute variables. (see figure 4.31)

4.5.5 True-False highlighting

The context menu includes settings for colored and black/white highlighting of tree nodes. (see figure 4.32) Enabling highlighting changes the color resp. font of the tree nodes. Nodes representing an expression, which evaluates to *true*, receives a green color resp. a bold font, if the No colors option is switched on. If an expression evaluates to *false*, the node appears red resp. inverse colored. Neutral nodes showing value assignments and undefined expressions are not highlighted. You can choose between different highlighting modes. They are listed below.

No Highlighting

Deactivates the highlighting.
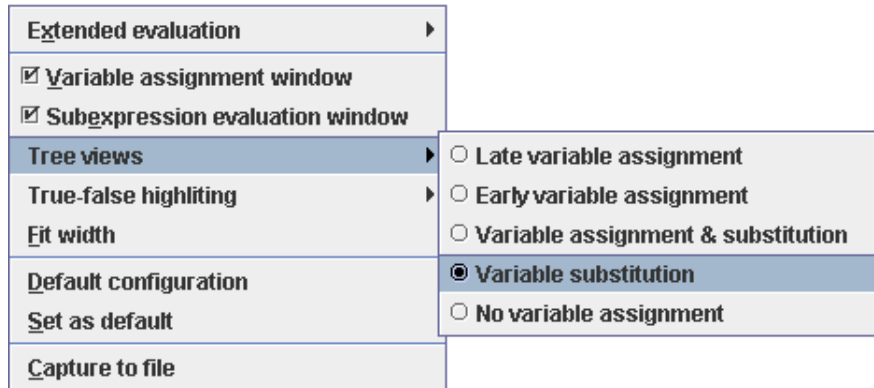
Figure 4.26: Menu - Tree Views

Term Highlighting

Highlights nodes of boolean type. (see figure 4.33)

Subtree Highlighting

Highlights nodes of boolean type and its child nodes if they have a different type. Child nodes, which also have a boolean type, receive a color depending on their truth value. The same highlighting rules are applied to their children. (see figure 4.34)

Complete Subtree Highlighting

The truth values of the immediate child nodes of the root specify the color of their subtrees. (see figure 4.35) Other nodes have no influence on the subtree colors.

4.5.6 Fit Width

This command fits the width of the OCL expressions to the width of the Evaluation Browser Window. You do not have to scroll horizontally.

4.5.7 Default Configuration

Restores the settings of the default configuration. The command `Set as default` stores these settings. (see section 4.5.8) There is a USE configuration file `etc/use.properties` where you can specify properties for all users. You can also create a local `.userc` file in your home directory, which overwrites or extends these settings. An example is shown below.

```
#Extended Evaluation Defaults
use.gui.view.evalbrowser.exists=false
use.gui.view.evalbrowser.forall=false
use.gui.view.evalbrowser.and=false
```
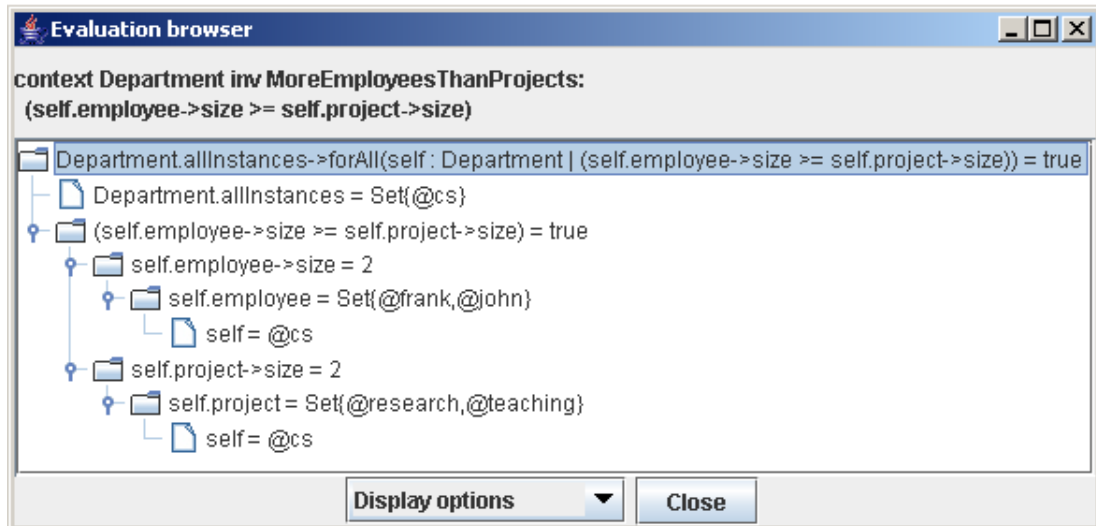
Figure 4.27: Evaluation Browser - Late Variable Assignment (Employees, Departments and Projects Example)

```
use.gui.view.evalbrowser.or=false
use.gui.view.evalbrowser.implies=false
#Extra-Windows
use.gui.view.evalbrowser.VarAssignmentWindow=false
use.gui.view.evalbrowser.SubExprSubstitutionWindow=false
#Tree-View Default
use.gui.view.evalbrowser.treeview=lateVarAssignment
#use.gui.view.evalbrowser.treeview=earlyVarAssignment
#use.gui.view.evalbrowser.treeview=substituteVarAssignment
#use.gui.view.evalbrowser.treeview=VarSubstitution
#use.gui.view.evalbrowser.treeview=noVarAssignment
#Highliting-Default
use.gui.view.evalbrowser.highliting=no
#use.gui.view.evalbrowser.highliting=term
#use.gui.view.evalbrowser.highliting=subtree
#use.gui.view.evalbrowser.highliting=complete
use.gui.view.evalbrowser.blackHighliting=false
```

4.5.8 Set to default

This commands opens the dialog shown in figure 4.36.

64
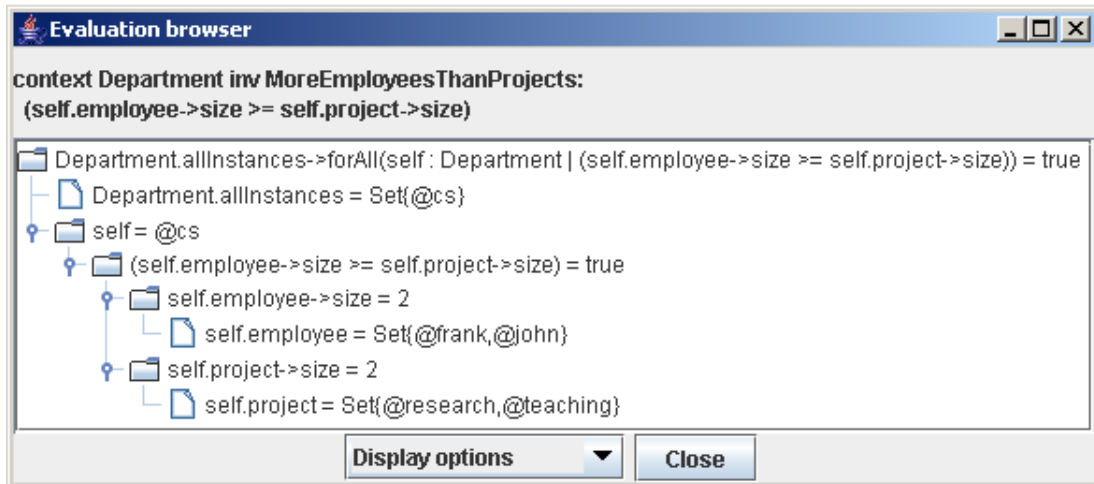
Figure 4.28: Evaluation Browser - Early Variable Assignment (Employees, Departments and Projects Example)

For This session

The current properties of the Evaluation Browser are saved, but not permanently. They are saved as long as the USE process is running.

For all sessions

The settings are saved permanently by changing the `.userc` file in your home directory.

Cancel

The dialog is closed without saving the actual configuration.

4.5.9 Capture to File

You can save the actual Evaluation Browser Window to file. This command opens a save dialog where you can define the destination directory, the filename and its format (PNG, JPG or BMP). PNG is the standard format.

4.5.10 Shortcuts

The above mentioned commands may be called by using shortcuts. The following table shows the shortcuts and the corresponding commands.
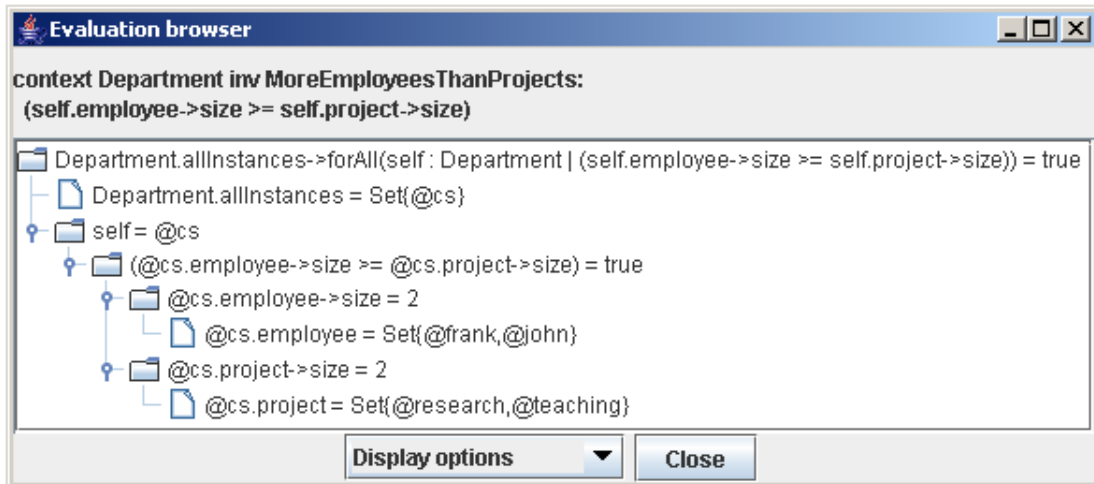
Figure 4.29: Evaluation Browser - Variable Assignment & Substitution (Employees, Departments and Projects Example)

Shortcut	Command
Alt-x	Extended Evaluation (all)
Alt-v	Variable Assignment Window
Alt-e	Subexpression Evaluation Window
Alt-1..4	Treeviews 1..4
Alt-0	disable Highlighting
Alt-7..9	Highlightings 1..3
Alt-f	Fit width
Alt-d	Default configuration
Alt-s	Set as default
Alt-c	Capture to file

4.5.11 Context Menu

You can click right on a tree node. If the node is closed, the context menu 4.37 is displayed. Use the Expand function to open the node. The Expand all command opens the complete subtree where the current node appears as root. The command Copy copies the OCL expression of the marked node to clipboard.
The context menu changes, if the selected node is opened. (see 4.38) The node may be closed by using the Collapse function. But this command does not close the child nodes. If you would like to close all child nodes too, use the Collapse all function instead.

4.5.12 Tree Display Menu

The Tree Display Menu is a drop down menu shown in figure 4.39. It is located on the left side of the close button. The menu entries are listed below.
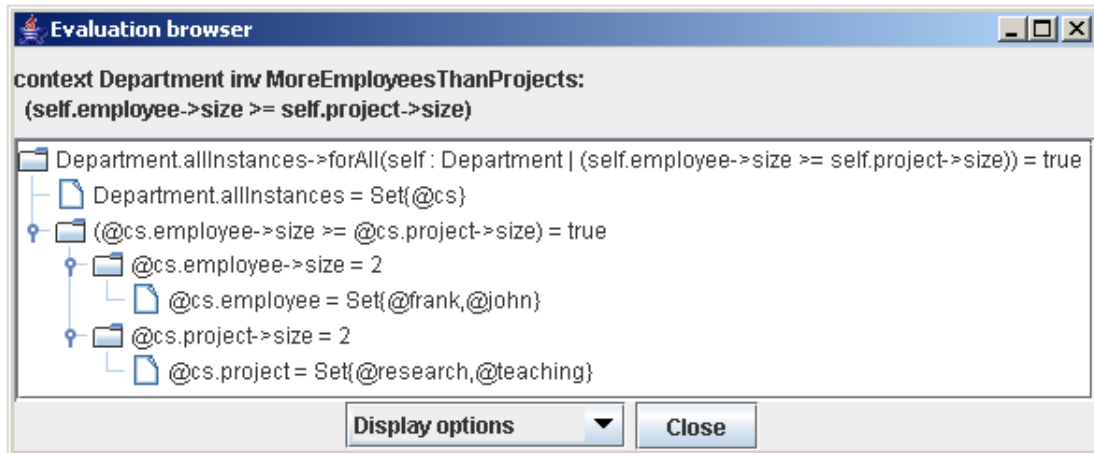
Figure 4.30: Evaluation Browser - Variable Substitution (Employees, Departments and Projects Example)

Expand all

Opens all nodes existing in the evaluation tree.

Expand all true

All displayed nodes of type boolean and their child nodes are opened if they evaluate to *true*.

Expand all false

All displayed nodes of type boolean and their child nodes are opened if they evaluate to *false*.

Collapse

This commands set the tree back to a state, where all nodes are folded up, but not the root.

4.5.13 Hide Title

The title of the Evaluation Browser shows the analyzed invariant and its definition. (see figure 4.40) If no invariant is actually analyzed, the expression of the root node is shown in the title. You can hide the title by double click on it. If it is hidden you can double click at the top margin to display the title again.

4.5.14 Object Browser

The Object Browser shows objects of user defined types, the valuation of their attributes, their associations and the objects connected via links of the corresponding associations. Double click on an object in the Variable Assignment Window to open the Object Browser. (see section 4.5.2)
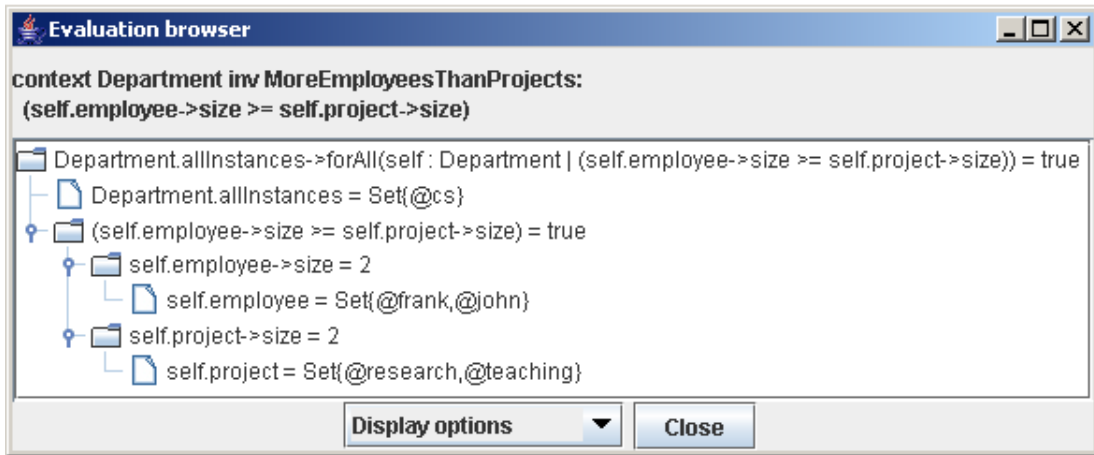
Figure 4.31: Evaluation Browser - No Variable Assignment (Employees, Departments and Projects Example)
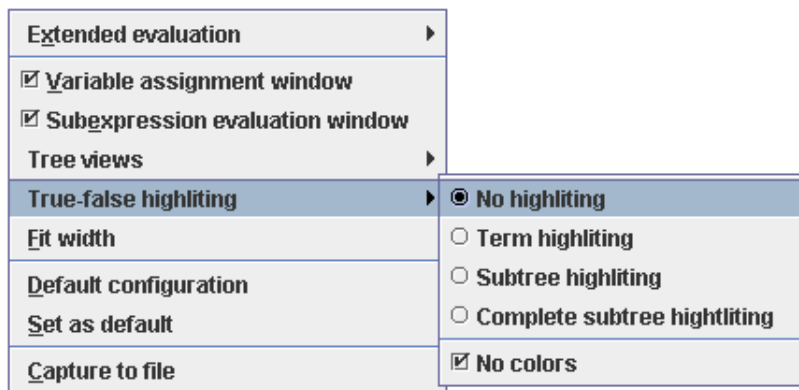
Figure 4.32: Menu - True False Highlighting

The figure 4.41 shows the browser listing the information about object @research in the Employees, Departments and Projects example. The first column shows its attributes, the second the corresponding values, the third shows its associations and the fourth shows the objects which are connected to @research via links. You can navigate to connected objects by choosing them in a drop down menu. Click left on the connected objects. This opens the menu, showing all objects reachable from the current object. After selecting the destination object, the Object Browser shows its properties. (see figure 4.42)
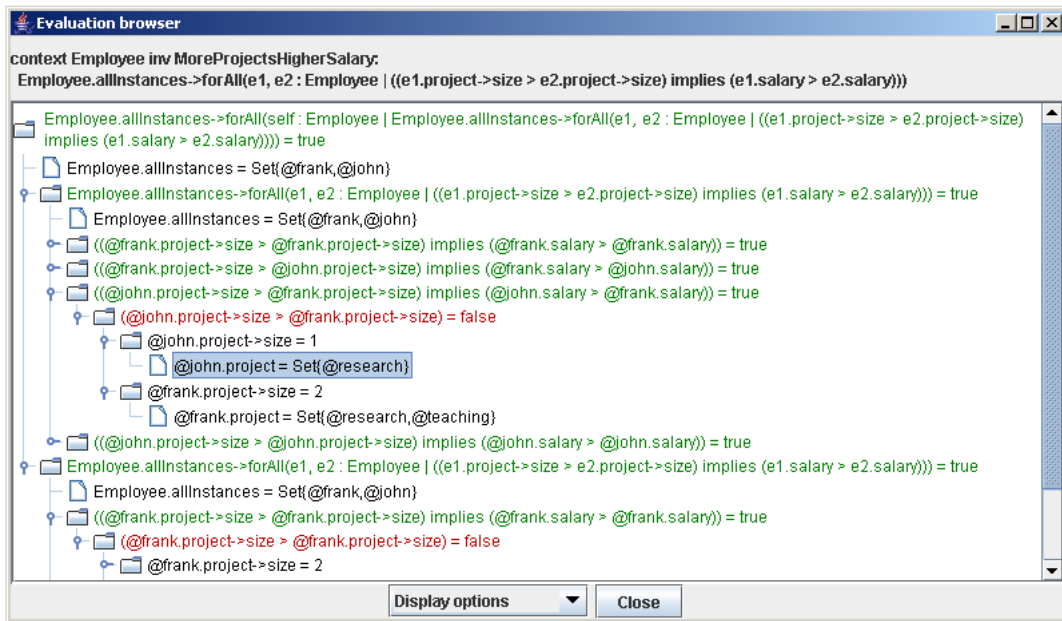
Figure 4.33: Evaluation Browser - Term Highlighting (Employees, Departments and Projects Example)
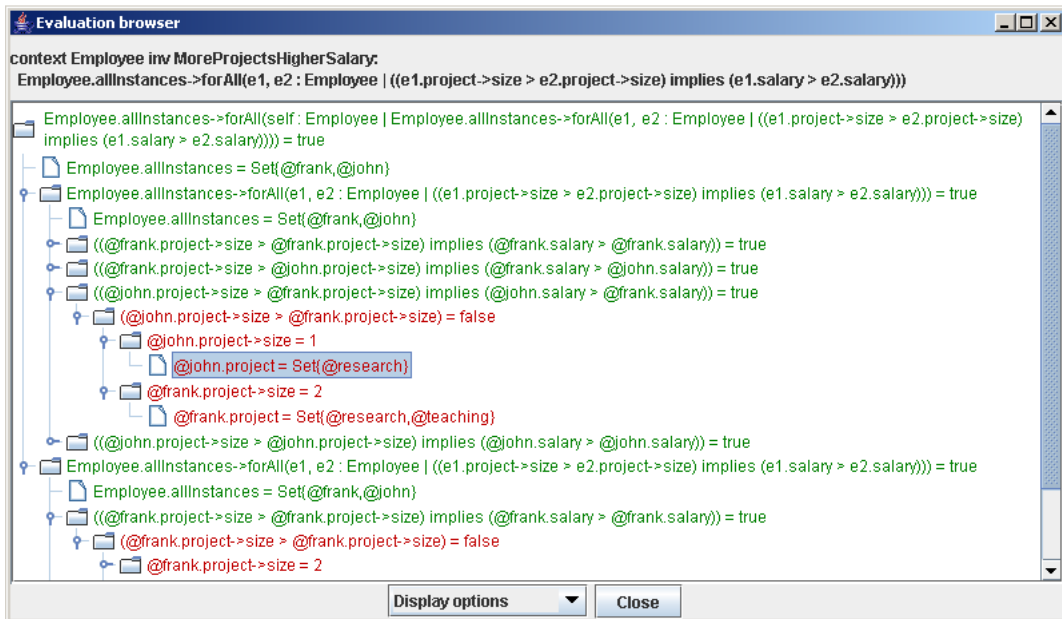
Figure 4.34: Evaluation Browser - Subtree Highlighting (Employees, Departments and Projects Example)
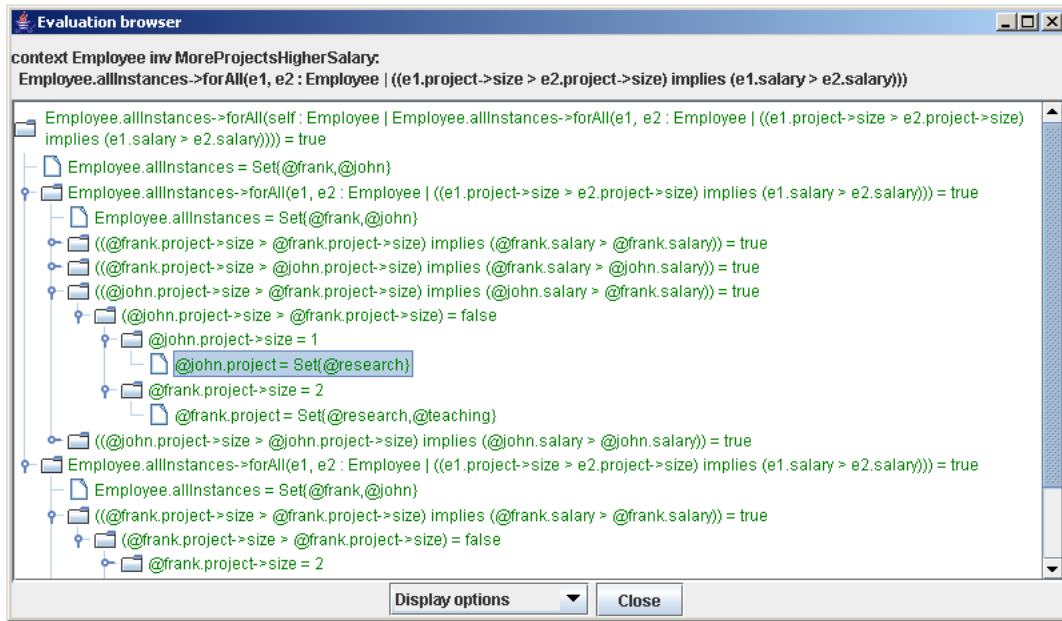
Figure 4.35: Evaluation Browser - Complete Subtree Highlighting (Employees, Departments and Projects Example)
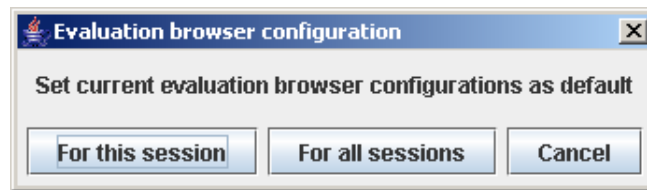
Figure 4.36: Evaluation Browser - Set as Default
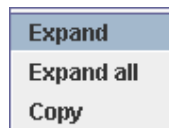
Figure 4.37: Evaluation Browser - Expand
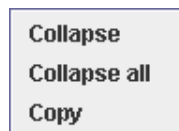
Figure 4.38: Evaluation Browser - Collapse

70
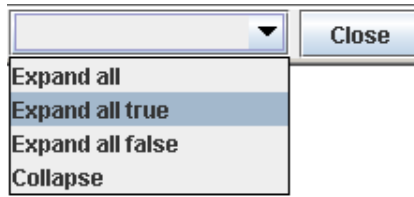
Figure 4.39: Evaluation Browser - Tree Display Menu and Close button

Figure 4.40: Evaluation Browser - Title
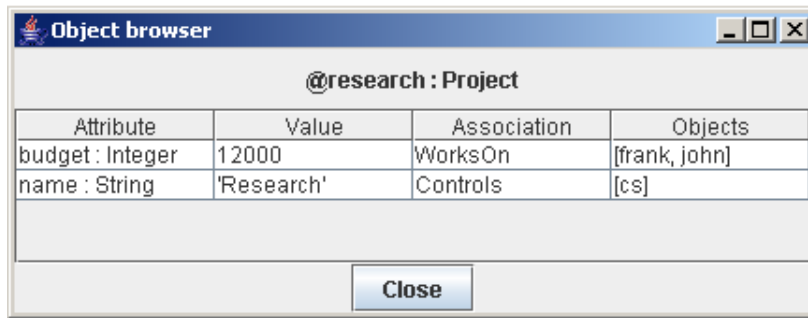
Figure 4.41: Evaluation Browser - Object Browser
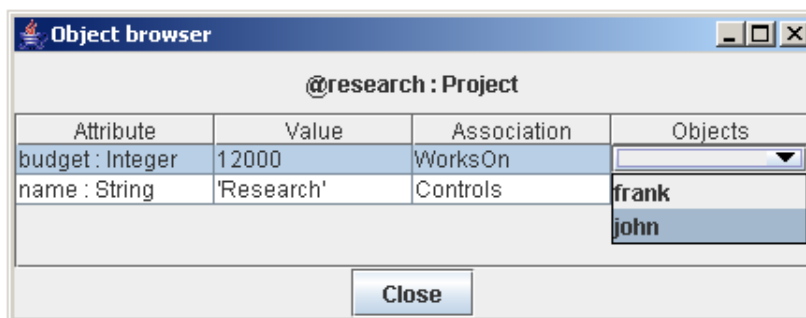
Figure 4.42: Evaluation Browser - Object Browser with Dropdown menu

71

5 Shell Reference

5.1 Commands

5.1.1 Overview of the Shell commands

Prints all available commands and a synopsis of their description.

Syntax: `help`

5.1.2 Help about a specific Shell command

Prints the syntax for the use of command *cmd* and its description with synopsis.

Syntax: `help` *cmd*

Example: `help !create`

5.1.3 Compile and evaluate an OCL expression

Compiles and evaluates the expression *OclExpr*. This Shell command is comparable to the function of the Evaluation Window in the GUI. (see section 4.1.3)

Syntax: `?` *OclExpr*

Example: The simple expression 2 = 2 has to be evaluated.

> **User input:** `? 2=2`

> **Result:** The result shown in the Shell.

> ```
> -> true : Boolean
> ```

5.1.4 Compile and evaluate an OCL expression (verbose)

Compiles and evaluates the Expression *OclExpr* with verbose output of subexpression results. After evaluating the expression the Evaluation Browser is displayed. It shows the evaluation tree for *OclExpr*.

Syntax: `??` *OclExpr*

Example: The simple expression 2 = 2 has to be evaluated in verbose mode.

> **User input:** `?? 2=2`

> **Result:** The result shown in the Shell.

```
Detailed results of subexpressions:
  2 : Integer = 2
  2 : Integer = 2
  (2 = 2) : Boolean = true
-> true : Boolean
```

5.1.5 Compile an OCL expression and show its static type

Compiles the expression *OclExpr* and shows its static type.

Syntax: : *OclExpr*

Example: The type of the expression 2 = 2 has to be identified.

    User input: : `2=2`

    Result: The result shown in the Shell.

```
-> Boolean
```

5.1.6 Enter OCL expressions over multiple lines

Use \ to enter a multiline mode. Finish with a '.' on a single line. In multiline mode an OCL expression may be split into several lines.

Syntax: The OCL expression is split into *n* parts. Lines may be blank.

```
\
OclExprPart1
OclExprPart2
...
OclExprPartn
.
```

Example: The Expression 2 = 2 is split into 3 lines. One of them is a blank line.

    User Input:

```
\
?2

=2
.
```

    Result: The result shown in the Shell.

```
-> true : Boolean
```

5.1.7 Create objects

Creates one or more objects of a given class or associationclass. The *newIdList* has to include at least one object name. These names identify the new objects of the type *class*. If *class* is an association class the link ends given with *idList* have to be specified with the keyword `between`. The order of the names in *idList* must conform to the definition of the associationclass.

Syntax: `!create` *newIdList* `:` *class* [`between` `(` *IdList* `)`]

Example for classes: The following commands create three objects for the class *Apple* and one for the classes *Lemon* and *Banana*.

> **User input:** The user input.
>
> ```
> !create greenApple, redApple, yellowApple : Apple
> !create bigLemon : Lemon
> !create banana : Banana
> ```

Example for association classes: This example creates an instance of the associationclass *FruitSalad*. The actual link ends are the existing objects *banana* with type *Banana* and *redApple* with type *Apple*.

> **User input:** The Shell command.
>
> ```
> !create smallSalad : FruitSalad between (banana, redApple)
> ```

5.1.8 Destroy objects

Destroys the objects given by the *idList*, which includes at least one object name. If the destroyed object is a link end, the corresponding link is deleted resp. the associationclass object is destroyed.

Syntax: `!destroy` *idList*

Example: The example destroys two objects.

> **User input:** `!destroy greenApple, smallSalad`

5.1.9 Insert a link into an association

Inserts a link between the objects in the *idList* into the association *assoc*.

Syntax: `!insert` *idList* `into` *assoc*

Example: This command inserts a link into the 3 ary association *Ingredients*. The second link end must have the type *Orange*. This is an abstract class. It cannot be instantiated. *Apple* is a subtype of this class. That means we may use the object *yellowApple*.

> **User input:** The Shell command.
>
> ```
> !insert (redApple, yellowApple, bigLemon) into Ingredients
> ```

5.1.10 Delete a link from an association

Deletes the link between the objects in the *idList* from the association *assoc*.

Syntax: `!delete` *idList* `from` *assoc*

Example: This command deletes the link inserted in section 5.1.9.

   User input: The Shell command.

```
!delete (redApple, yellowApple, bigLemon) from Ingredients
```

5.1.11 Set an attribute value of an object

Sets the attribute *attr* of the object *obj* to a new value given by *OclExpr*.

Syntax: `!set` *obj.attr* `:=` *OclExpr*

Example: Both commands set the boolean attribute of object *redApple* to true. It inherits the *juice* attribute from *Orange*.

   User input: Two shell commands.

```
!set redApple.juice := 2=2
!set redApple.juice := true
```

5.1.12 Enter object operation

Invokes an operation with the name *OpName* on the object *ObjExpr*. If the operation has *n* parameters, the *ExprList* includes *n* expressions which evaluate to the corresponding values. If there is more than one operation call a call stack is used to remember the entered operations. The deepest call has to be exited first.

Syntax: `!openter` *ObjExpr OpName(ExprList)*

Example without parameters: The object *banana* is an instance of class *Banana* which defines the operation *peel*. This operation has no parameters.

   User input: `!openter banana peel()`

   Result: The preconditions are checked.

```
precondition 'pre1' is true
```

Example with formal parameter: The operation *squeeze* is invoked on object *bigLemon*. It has one explicitly defined parameter of type *Integer*.

   User input: `!openter bigLemon squeeze(11)`

   Result: The second precondition is false because the argument is greater than 10. The operation call is canceled.

```
precondition 'pre2' is true
precondition 'lessThanTenOranges' is false
```

User input: `!openter bigLemon squeeze(3)`

Result: Both preconditions are true.

```
precondition 'pre2' is true
precondition 'lessThanTenOranges' is true
```

5.1.13 Exit least recently entered operation

This command exits the least recently entered operation, i.e. the top of the call stack. If the operation has a return value it has to be specified behind the `opexit` command.

Syntax: `!opexit` *ReturnValExpr*

Example - second call: The operation *squeeze* is the least recently entered operation. Its return value has to be of type *Integer*.

> **User input:** `!opexit 20`

> **Result:** The postconditions are checked.

> ```
> postcondition 'alwaysTrue' is true
> ```

Example - first call: The operation *peel* has been called before. It can be exited now. The result value has to be a *String*.

> **User input:** `!opexit 'failure'`

> **Result:** One postcondition is false because the result value has to be *'theResult'*. Even though the result value is wrong the operation is exited.

> ```
> postcondition 'post1' is true
> postcondition 'post2' is false
> evaluation results:
>   result : String = 'failure'
>   'theResult' : String = 'theResult'
>   (result = 'theResult') : Boolean = false
> ```

> **User input:** `!openter 'theResult'`

> **Result:** If we exit this operation with the right result value both preconditions appear true.

> ```
> postcondition 'post1' is true
> postcondition 'post2' is true
> ```

5.1.14 Check integrity constraints

The command checks the structure, i.e. the multiplicity constraints and the invariant constraints. There are four optional parameters. `-v` enables the verbose output of the subexpression results for violated invariants. The option `-d` shows which instances cause an invariant to fail. The option `-a` checks all invariants including the ones loaded by the generator. You can specify which invariants should be checked by entering an *invList*. Use the following invariant signature: *context*`::`*invName*. The signatures have to be separated with a blank.

Syntax: `check` [-v] [-d] [-a | *invList*]

Example without options:

> **User input:** `check`

> **Result:** The result shows several multiplicity constraint violations, because the example script does not create enough links. One invariant is violated in the current system state.

```
checking structure...
Multiplicity constraint violation in association 'AppleSpritzer':
  Object 'yellowApple' of class 'Apple'
    is connected to 0 objects of class 'Lemon' via role 'flavor'
  but the multiplicity is specified as '1..8,10,15..*'.
...
Multiplicity constraint violation in association 'Ingredients':
  Objects 'yellowApple, redApple'
    are connected to 0 objects of class 'Lemon'
  but the multiplicity is specified as '1..*'.
checking invariants...
checking invariant (1) 'Orange::OrangeInv': OK.
checking invariant (2) 'Orange::alwaysTrue': OK.
checking invariant (3) 'Orange::inv2': FAILED.
  -> false : Boolean
checking invariant (4) 'Peach::inv1': OK.
checking invariant (5) 'Peach::neverViolated': OK.
checked 5 invariants in 0.046s, 1 failure.
```

Example with options:

> **User input:** `check -d`

> **Result:** The option -d shows, that *redApple* and *yellowApple* violate the invariant *inv2*. with these options.

```
...
checking invariant (3) 'Orange::inv2': FAILED.
  -> false : Boolean
Instances of Orange violating the invariant:
  -> Set{@redApple,@yellowApple} : Set(Apple)
...
```

5.1.15 Activate single-step mode

This command activates the single step mode to read in a script step by step.

Syntax: `step on`

Example: Read in the script.

User input: Enter the step on mode and open the script.

```
use> step on
Step mode turned on.
use> open ../examples/Documentation/ExampleSpecification/ExampleScript.cmd
[step mode: 'return' continues, 'escape' followed by 'return'
exits step mode.]
```

Result: Now you can press return to read in the script command by command.

5.1.16 Read information from File

Reads information from a file. It may be a USE specification (*fileName*`.use`), a command file (*fileName*`.cmd`), or an invariants file (*fileName*`.invs`). If a command file is read in, every line is shown in the Shell. You have to load a specification before you can read in command files. The `-q` option allows a quiet reading. If the filename is in the root directory of USE, there is no need to enter the path. If the file exists in a subdirectory of the USE root directory (usually named `use`-*version*), you have to enter the sub path beginning at the USE root. (see example) If the file does not exist in the USE directory you have to enter the whole path.

Syntax: open [*path*] *fileName*`.(` use `|` cmd `|` invs `)`

Example file in USE sub directory: Read in a specification existing in a subdirectory of USE.

User input: `open ../examples/Documentation/Demo/Demo.use`

Result:

```
compiling specification...
Model Company (3 classes, 3 associations, 4 invariants,
0 operations, 0 pre-/postconditions)
```

Example file not in USE directory: Read in a specification.

User input: `open /home/opti/ExampleSpecification.use`

Result:

```
compiling specification...
Model Fruits (6 classes, 3 associations, 5 invariants,
3 operations, 6 pre-/postconditions)
```

5.1.17 Reset system to empty state

Resets the USE system state to an empty state. All objects and links are deleted.

Syntax: `reset`

5.1.18 Exit USE

Enter `q`, `quit` or `exit` to exit USE.

Syntax: `(` q `|` quit `|` exit `)`

5.1.19 Undo last state manipulation command

Undoes the last state manipulation command.

Syntax: `undo`

5.1.20 Print info about a class

Prints information about a class existing in the specification.

Syntax: `info class` *className*

Example: Get information about the *Apple*.

User input: `info class Apple`

Result: The result shown in the Shell.

```
class Apple < Lemon,Orange
end
2 objects of this class in current state.
```

5.1.21 Print info about loaded model

Prints all information about the loaded model (classes, associations, constraints).

Syntax: `info model`

Example: Information about the example model.

User input: `info model`

5.1.22 Print info about current system state

Prints information about the current system state. Shows how many objects and links are created.

Syntax: `info state`

Example: Information about the current state.

User input: `info state`

Result: The result shown in the Shell.

```
State: state#4
class       : #objects + #objects in subclasses
---------------------------------------------
Apple       :         2                    2
Banana      :         1                    1
FruitSalad  :         0                    0
Lemon       :         1                    4
(Orange)    :         0                    2
```

```
Peach        :         0                           0
------------------------------------------------
total        :         4

association  : #links
---------------------
AppleSpritzer :        0
FruitSalad    :        0
Ingredients   :        0
---------------------
total         :        0
```

5.1.23 Print currently active operations

Prints all operations, which did not terminate yet, i.e. the operation call stack.

Syntax: `info opstack`

Example: Invokes an operation and requests information about the operation stack.

User input:

```
use> !openter bigLemon squeeze(1)
precondition 'pre2' is true
precondition 'lessThanTenOranges' is true
use> info opstack
```

Result: The result shown in the Shell.

```
active operations:
1. Lemon::squeeze(i : Integer) : Integer | bigLemon.squeeze(1)
```

5.1.24 Print internal program info

Prints information about the USE process (i.e., memory usage).

Syntax: `info prog`

Example: User input: `info prog`

Result: The result shown in the Shell.

```
(mem: 27% = 793.800 bytes free, 2.916.352 bytes total)
```

5.1.25 Print information about global variables

Prints information about global variables.

Syntax: `info vars`

Example: The simple expression 2 = 2 has to be evaluated.

User input: `info vars`

Result: The result shown in the Shell.

```
redApple : Apple = @redApple
yellowApple : Apple = @yellowApple
bigLemon : Lemon = @bigLemon
banana : Banana = @banana
i : Integer = 1
self : Lemon = @bigLemon
```

5.2 Generator

Description will be available in the next document version.

6 OCL Standard Operations

The OCL Standard operations are described following [Kya06].

6.1 Object Types

6.1.1 Equality

$= (y : OclAny) : Boolean$ represents equality between objects. It evaluates to true if *self* is the same as *y*.
Notation: $self = y$

6.1.2 Inequality

$<> (y : OclAny) : Boolean \stackrel{def}{=} not (self = y)$ represents inequality between objects.
Notation: $self <> y$

6.1.3 isUndefined

isUndefined() : *Boolean* evaluates to true, if the callee is undefined.
Notation: $self$ `.isUndefined()`

6.1.4 oclIsNew

$oclIsNew() : Boolean \stackrel{def}{=} self@pre.isUndefined()$ can only be used in a postcondition and states that the object has been newly created during the execution of an operation.
Notation: $self$ `.oclIsNew()`

6.1.5 oclAsType

$oclAsType(T) : T$ is a "cast" or "retyping" expression, evaluating to the value of the callee, if it is an instance of T, and to *Undefined* otherwise.
Notation: $self$ `.oclAsType(`T`)`

6.1.6 oclIsTypeOf

$oclIsTypeOf(T) : Boolean$ evaluates to *true* if the callee is an instance of type T.
Notation: $self$ `.oclIsTypeOf(`T`)`

6.1.7 oclIsKindOf

oclIsKindOf(T) : *Boolean* evaluates to *true* if the callee is an instance of type T or one of Ts subtypes, that is, the callee conforms to the type T.
Notation: *self* .isKindOf(T)

6.2 Boolean Types

a	b	not b	a and b	a or b	a implies b	a xor b
true	true	false	true	true	true	false
true	false	true	false	true	false	true
true	⊥	⊥	⊥	true	⊥	⊥
false	true	false	false	true	true	true
false	false	true	false	false	true	false
false	⊥	⊥	false	⊥	true	⊥
⊥	true	false	⊥	true	true	⊥
⊥	false	true	false	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥

6.3 Real

6.3.1 Addition

$+$(y : *Real*) : *Real* describes the sum of *self* and y.
Notation: *self*$+y$

6.3.2 Subtraction

$-$(y : *Real*) : *Real* describes the difference between *self* and y.
Notation: *self*$-y$

6.3.3 Multiplication

$*$(y : *Real*) : *Real* describes the product of *self* and y.
Notation: *self*$*y$

6.3.4 Division

$/$(y : *Real*) : *Real* describes the quotient of *self* and y.
Notation: *self*$/y$

6.3.5 Negation

$-$() : *Real* $\overset{def}{=} 0 - self$ describes the negation of *self*.
Notation: $-self$

6.3.6 Less

$<$ (*y* : *Real*) : *Boolean* evaluates to *true*, if the value of *self* is less than the value of *y*. It evaluates to *false*, if the value of *self* is equal to or greater than the value of *y*. In any other case, it is undefined.
Notation: *self* $<y$

6.3.7 Greater

$>$ (*y* : *Real*) : *Boolean* evaluates to *true*, if the value of *self* is greater than the value of *y*. It evaluates to *false*, if the value of *self* is less than or equal to the value of *y*. In any other case, it is undefined.
Notation: *self* $>y$

6.3.8 Less or equal

$<=$ (*y* : *Real*) : *Boolean* evaluates to *true*, if the value of *self* is less than or equal to the value of *y*. It evaluates to *false*, if the value of *self* is greater than the value of *y*. In any other case, it is undefined.
Notation: *self* $<=y$

6.3.9 Greater or equal

$>=$ (*y* : *Real*) : *Boolean* evaluates to *true*, if the value of *self* is equal to or greater than the value of *y*. It evaluates to *false*, if the value of *self* is less than the value of *y*. In any other case, it is undefined.
Notation: *self* $>=y$

6.3.10 Absolute Values

abs() : *Real* $\stackrel{def}{=}$ *if self* < 0 *then* $-$ *self else self endif* describes the absolute value of *self*.
Notation: *self* `.abs()`

6.3.11 Floor

floor() : *Integer* describes the largest integer which is not greater than *self*.
Notation: *self* `.floor()`
Note: *floor* binds stronger than $-$. That means $(-3.3).floor() = -4$ and $-3.3.floor() = -(3.3.floor()) = -3$.

6.3.12 Round

round() : *integer* $\stackrel{def}{=}$ $(self + 0.5).floor()$ rounds *self* to the nearest integer.
Notation: *self* `.round()`

6.3.13 Maximum

$max(y : Real) : Real \stackrel{def}{=}$ *if self $< y$ theny else self endif* results in the greater value of *self* and *y*.
Notation: *self*.`max`(*y*)

6.3.14 Minimum

$min(y : Real) : Real \stackrel{def}{=}$ *if self $> y$ then y else self endif* results in the smaller value of *self* and *y*.
Notation: *self*.`min`(*y*)

6.4 Integer

6.4.1 Addition

$+(y : Integer) : Integer$ describes the sum of *self* and *y*.
Notation: *self*+*y*

6.4.2 Subtraction

$-(y : Integer) : Integer$ describes the difference between *self* and *y*.
Notation: *self*-*y*

6.4.3 Multiplication

$*(y : Integer) : Integer$ describes the product of *self* and *y*.
Notation: *self**y

6.4.4 Division

$/(y : Integer) : Real$ describes the quotient of *self* and *y*.
Notation: *self*/*y*

6.4.5 Negation

$-() : Integer \stackrel{def}{=} 0 - self$ describes the negation of *self*.
Notation: -*self*

6.4.6 Less

$< (y : Integer) : Boolean$ evaluates to *true*, if the value of *self* is less than the value of *y*. It evaluates to *false*, if the value of *self* is equal to or greater than the value of *y*. In any other case, it is undefined.
Notation: *self*<*y*

6.4.7 Greater

$>$ (y : *Integer*) : *Boolean* evaluates to *true*, if the value of *self* is greater than the value of y. It evaluates to *false*, if the value of *self* is less than or equal to the value of y. In any other case, it is undefined.
Notation: *self>y*

6.4.8 Less or equal

$<=$ (y : *Integer*) : *Boolean* evaluates to *true*, if the value of *self* is less than or equal to the value of y. It evaluates to *false*, if the value of *self* is greater than the value of y. In any other case, it is undefined.
Notation: *self<=y*

6.4.9 Greater or equal

$>=$ (y : *Integer*) : *Boolean* evaluates to *true*, if the value of *self* is equal to or greater than the value of y. It evaluates to *false*, if the value of *self* is less than the value of y. In any other case, it is undefined.
Notation: *self>=y*

6.4.10 Absolute Values

abs() : *Integer* $\stackrel{def}{=}$ *if self* < 0 *then* $-$ *self else self endif* describes the absolute value of *self*.
Notation: *self*.abs()

6.4.11 Euclidean division

div(y : *Integer*) : *Integer* describes Euclidean division of *self* by y, that is, it results in the unique integer z such that there exists an $0 \leq r < y$ with $z * y + r = self$.
Notation: *self* div *y*

6.4.12 Modulo

mod(y : *Integer*) : *Integer* describes Euclidean reminder of *self* divided by y, that is, it results in the unique integer $0 \leq r < y$ such that there exists an integer z with $z * y + r = self$.
Notation: *self*.mod(*y*)

6.4.13 Maximum

max(y : *Integer*) : *Integer* $\stackrel{def}{=}$ *if self* $< y$ *then y else self endif* evaluates to the greater value of *self* and y.
Notation: *self*.max(*y*)

6.4.14 Minimum

$min(y : Integer) : Integer \overset{def}{=} if\ self > y\ then\ y\ else\ self\ endif$ evaluates to the smaller value of *self* and *y*.
Notation: $self.\texttt{min}(y)$

6.5 Collection

$\langle col \rangle ::= Set \mid Bag \mid Sequence$

6.5.1 Size

$size() : Integer \overset{def}{=} self \rightarrow iterate(e; a : Integer = 0 \mid a + 1)$
Notation: $self\texttt{->size()}$

6.5.2 Count

$count(, y : T) : Integer \overset{def}{=} self \rightarrow iterate(i; a : Iterate = 0 \mid if\ y = i\ then\ a + 1\ else\ a\ endif)$
counts how often *y* occurs in the collection.
Notation: $self\texttt{->count}(y)$

6.5.3 Includes

$includes(, y : T) : Boolean \overset{def}{=} self \rightarrow count(y) > 0$ returns true if and only if *y* occurs in the collection.
Notation: $self\texttt{->includes}(y)$

6.5.4 Excludes

$excludes(, y : T) : Boolean \overset{def}{=} self \rightarrow count(y) = 0$ returns true if and only if *y* does not occur in the collection.
Notation: $self\texttt{->excludes}(y)$

6.5.5 Includes all

$includesAll(, y : \langle col \rangle(T)) : Boolean \overset{def}{=} y \rightarrow forAll(e \mid self \rightarrow includes(e)).$
Notation: $self\texttt{->includesAll}(y)$

6.5.6 Excludes all

$excludesAll(, y : \langle col \rangle(T)) : Boolean \overset{def}{=} y \rightarrow forAll(e \mid self \rightarrow excludes(e)).$
Notation: $self\texttt{->excludesAll}(y)$

6.5.7 Is empty

$isEmpty() : Boolean \stackrel{def}{=} self \rightarrow size() = 0.$
Notation: $self$->isEmpty()

6.5.8 Not empty

$notEmpty() : Boolean \stackrel{def}{=} self \rightarrow size() <> 0.$
Notation: $self$->notEmpty()

6.5.9 Sum

$sum() : T \stackrel{def}{=} self \rightarrow iterate(e; a : Iterate = 0 \mid a + e).$
Notation: $self$->sum()

6.6 Set

6.6.1 Set-Equality

$= (y : Set(T)) : Boolean$ describes set-equality.
Notation: $self=y$

6.6.2 Including elements

$including(y : T) : Set(T)$ describes the set obtained from $self$ by including y.
Notation: $self$->including(y)

6.6.3 Excluding elements

$excluding(y : T) : Set(T)$ describes the set obtained from $self$ by excluding y.
Notation: $self$->excluding(y)

6.6.4 Union

$union(y : Set(T)) : Set(T)$ describes the union of the set $self$ and the set y.
Notation: $self$->union(y)

6.6.5 Union with Bag

$union(y : Bag(T)) : Bag(T)$ describes the union of the set obtained from $self$ by assuming that each element of $self$ occurs exactly once and the bag y.
Notation: $self$->union(y)

6.6.6 Intersection

intersection(*y* : *Set*(*T*)) : *Set*(*T*) describes the intersection of the set *self* the set *y*.
Notation: *self* ->intersection(*y*)

6.6.7 Intersection with Bag

intersection(*y* : *Bag*(*T*)) : *Set*(*T*) describes the intersection of the set *self* and the set obtained from *y* by including every element contained in *y*.
Notation: *self* ->intersection(*y*)

6.6.8 Difference of sets

−(*y* : *Set*(*T*)) : *Set*(*T*) describes the difference of the set *self* the set *y*.
Notation: *self* -*y*

6.6.9 Flatten

flatten() : *Set*(*T*′). If *self* is a set of collections, then this operation returns the set-union of all its elements.
Notation: *self* ->flatten()

6.6.10 As Bag

asBag() : *Bag*(*T*) $\stackrel{def}{=}$ *self* → *iterate*(*e*; *a* : *Bag*(*T*) = *oclEmpty*(*Bag*(*T*)) | *a* → *including*(*e*)) returns a bag which includes each element of *self* exactly once.
Notation: *self* ->asBag()

6.6.11 As Sequence

asSequence() : *Sequence*(*T*) returns a sequence containing all elements of *self* exactly once. The order of the elements is arbitrary. It is equivalent to the expression

$$self → iterate(e; a : Sequence(T) = oclEmpty(Sequence(T) | a → append(e)) \quad .$$

Notation: *self* ->asSequence()

6.7 Bag

6.7.1 Equality

= (*y* : *Bag*(*T*)) : *Boolean* describes equality of multi-sets.
Notation: *self* =*y*

6.7.2 Including elements

including(*y* : *T*) : *Bag*(*T*) describes the bag obtained from *self* by including *y*.
Notation: *self* ->including(*y*)

6.7.3 Excluding elements

excluding(*y* : *T*) : *Bag*(*T*) describes the bag obtained from *self* by excluding all occurrences of *y*.
Notation: *self* ->excluding(*y*)

6.7.4 Union

union(*y* : *Bag*(*T*)) : *Bag*(*T*) describes the union of the bag *self* and the bag *y*.
Notation: *self* ->union(*y*)

6.7.5 Union with Set

union(*y* : *Set*(*T*)) : *Bag*(*T*) describes the union of the bag *self* and the set obtained from *y* by including each element of *y* exactly once.
Notation: *self* ->union(*y*)

6.7.6 Intersection

intersection(*y* : *Bag*(*T*)) : *Bag*(*T*) describes the intersection of the bag *self* and the bag *y*.
Notation: *self* ->intersection(*y*)

6.7.7 Intersection with Set

intersection(*y* : *Set*(*T*)) : *Set*(*T*) describes the intersection of the bag *self* and the set *y*.
Notation: *self* ->intersection(*y*)

6.7.8 Flatten

flatten() : *Bag*(*T'*). If *self* is a bag of collections, then this operation returns the bag union of all its elements.
Notation: *self* ->flatten(*y*)

6.7.9 As Set

$asSet() : Set(T) \stackrel{def}{=} self \rightarrow iterate(e; a : Set(T) = oclEmpty(Set(T)) \mid a \rightarrow including(e))$ returns a set which contains each element of *self*.
Notation: *self* ->asSet()

6.7.10 As Sequence

asSequence() : *Sequence*(*T*) returns a sequence containing all elements of *self* as often as they occur in the multi-set. The order of the elements is arbitrary. It is equivalent to:

$$self \rightarrow iterate(e; a : Sequence(T) = oclEmpty(Sequence(T)) \mid a \rightarrow append(e)) \quad .$$

Notation: *self* ->union(*y*)

6.8 Sequence

6.8.1 Get element

at(*y* : *Integer*) : *T* results in the element at the *y*th position of the sequence.
Notation: *self* ->at(*y*)

6.8.2 Equality

= (*y* : *Sequence*(*T*)) : *Boolean* $\overset{def}{=}$ *let s = self* → *size*() *in s = y* → *size*() *and Sequence*{1..*s*} → *forAll*(*i* : *Integer* | *self* → *at*(*i*) = *y* → *at*(*i*)) describes equality of sequences.
Notation: *self*=*y*

6.8.3 Union

union(*y* : *Sequence*(*T*)) : *Sequence*(*T*) describes the concatenation of *self* and *y*.
Notation: *self* ->union(*y*)

6.8.4 Flatten

flatten(*self* : *Sequence*(*T*)) : *Sequence*(*T*′). If *self* is a sequence of collections, then this operation returns the sequence concatenation of all its elements.
Notation: *self* ->flatten()

6.8.5 Append elements

append(*y* : *T*) : *Sequence*(*T*) $\overset{def}{=}$ *self* → *union*(*Sequence*{*y*}) results in the sequence which consists of all elements of *y* with *y* appended.
Notation: *self* ->append(*y*)

6.8.6 Prepend elements

prepend(*y* : *T*) : *Sequence*(*T*) $\overset{def}{=}$ *Sequence*{*y*} → *union*(*self*) results in the sequence which consists of all elements of *self* with *y* prepended.
Notation: *self* ->prepend(*y*)

6.8.7 Excluding elements

excluding(*y* : *T*) : *Sequence*(*T*) $\stackrel{def}{=}$ *self* → *iterate*(*i*; *a* : *Sequence*(*T*) = *oclEmpty*(*Sequence*(*T*)) | *if* *y* = *i* *then* *a* *else* *a* → *append*(*i*) *endif*) results in the largest sub-sequence of *self*, in which *y* does not occur.
Notation: *self* ->excluding(*y*)

6.8.8 Subsequence

subSequence(*y* : *Integer*, *z* : *Integer*) : *Sequence*(*T*) results in the subsequence of *self* starting at index *y* and ending at index *z*. It is equivalent to:

$$Sequence\{1..z\} \rightarrow Iterate(i; a : Sequence(T) = oclEmpty(Sequence(T)) \ |$$
$$if \ y \leq i \ and \ i \leq z \ then \ a \rightarrow append(self \rightarrow at(i))else \ a \ endif)$$

Notation: *self* ->subSequence(*y*)

6.8.9 Get first element

first() : *T* $\stackrel{def}{=}$ *self* → *at*(1).
Notation: *self* ->first(*y*)

6.8.10 Get last element

last() : *T* $\stackrel{def}{=}$ *self* → *at*(*self* → *size*()).
Notation: *self* ->last(*y*)

6.8.11 As Set

asSet() : *asSet*(*T*) $\stackrel{def}{=}$ *self* → *iterate*(*e*; *a* : *Set*(*T*) = *oclEmpty*(*Set*(*T*)) | *a* → *including*(*e*)) returns a set which contains each element of *self*.
Notation: *self* ->asSet(*y*)

6.8.12 As Bag

asBag() : *Bag*(*T*) $\stackrel{def}{=}$ *self* → *iterate*(*e*; *a* : *Bag*(*T*) = *oclEmpty*(*Bag*(*T*)) | *a* → *including*(*e*)) returns a bag containing all elements of the sequence *self*.
Notation: *self* ->asBag(*y*)

Bibliography

[GR98a]   Martin Gogolla and Mark Richters.   On constraints and queries in UML.   In Martin Schader and Axel Korthaus, editors, *The Unified Modeling Language - Technical Aspects and Applications*, pages 109–121, Heidelberg, 1998. Physica-Verlag.

[GR98b]   Martin Gogolla and Mark Richters. On formalizing the uml object constraint language OCL. In Tok Wang Ling, Sudha Ram, and Mong Li Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, volume 1507 of *LNCS*, pages 449–464, Berlin, 1998. Springer.

[GR99]   Martin Gogolla and Mark Richters.   A metamodel for OCL.   In Robert France and Bernhard Rumpe, editors, *Proceedings of the Second International Conference on the Unified Modeling Language: UML'99*, volume 1723 of *LNCS*. Springer, 1999.

[GR00]   Martin Gogolla and Mark Richters. Validating UML models and OCL constraints. In *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference*, volume 1939 of *LNCS*, York, UK, 2000. Springer.

[Kya06]   Marcel Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Lehmanns Media, Berlin, 2006.

[Obj99]   Object Management Group, Inc.   *OMG Unified Modeling Language Specification, Version 1.3*, June 1999. Internet: http://www.omg.org.

[Ric02]   Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*, volume 14 of *BISS Monographs*. Logos, Berlin, 2002.