Detecting OCL Traps in the UML 2.0 Superstructure An Experience Report

Hanna Bauerdick, Martin Gogolla, and Fabian Gutsche

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany
[hanna|gogolla|gutsche]@informatik.uni-bremen.de
http://www.db.informatik.uni-bremen.de

Abstract. Currently, the OMG is developing a new version of the Unified Modeling Language (UML), UML 2.0, which involves major innovations in its metamodel. As for previous versions of the UML, the Object Constraint Language (OCL) is employed to give restrictions on the use of UML and for the formulation of additional operations. It seems that the OCL expressions in the current version of the UML 2.0 Superstructure have not been checked with a tool. In this paper we report on an experiment in checking and validating the well-formedness rules and operation definitions of the UML 2.0 Superstructure w.r.t. syntax and type checking by using our tool USE (UML-based Specification Environment). For this purpose we classify the errors detected by USE in appropriate error categories. We develop statistical information on error frequencies w.r.t. package location and error category. All errors detected by USE and their detailed description are made available in a separate EXCEL file.

1 Introduction

Today, the Unified Modeling Language (UML) [5, 4] is commonly used as a description language within object oriented software development. For UML models, the Object Constraint Language (OCL) [5, 7] is often applied to describe details of a model more precisely.

In the near future, a new version of UML will be released, UML 2.0 [6]. Apart from two parts dealing with OCL and diagram interchange, this UML version consists of two complementary documents: The UML Infrastructure defines the elementary language constructs and therefore provides the basic architecture for the UML Superstructure, which allocates the user constructs. Both parts, the Infrastructure and the Superstructure, are described with UML class diagrams and OCL expressions. In order to get more insight into the use of OCL within the UML metamodel, we have analyzed and categorized the OCL expressions of UML 2.0 by using our OCL tool USE [3]. The result of this validation process is collected in a comprehensive EXCEL file [1] and shows that many OCL errors arise due to errors in the underlying class diagrams.

The aim of this study is first to give concrete hints to improve technical details of the UML 2.0, second to provide a larger case study showing the applicability of our tool USE, and third to make first steps for a test methodology for UML models with significant OCL parts and in particular to explore possibilities to document OCL errors appropriately.

The rest of the paper is structured as follows. Section 2 discusses extracts of the comprehensive validation, such as error categories which we found out to be adequate and expressive examples demonstrating typical problems which frequently occurred. In Sect. 3, statistical information about error frequencies will be presented. The paper closes with a short conclusion in Sect. 4.

2 OCL Traps in the UML 2.0 Superstructure

USE offers the possibility to check UML models on inconsistencies w.r.t. the class structure and OCL constraints. To validate the UML 2.0 Superstructure we have utilized this feature of USE. We have converted the superstructure MDL [2] file of the OMG into a USE compliant format to accomplish the validation process.

We identified the following five main error categories: Category E1 is for plain syntax errors; category E2 describes minor inconsistencies; category E3 classifies type checking errors; category E4 stands for general problems; category E5 registers inconsistencies with USE. Errors which occurred more often were parameterized and consequently they have their own sub-category. In the following we will explain the different categories and give typical examples.

2.1 Category E1: Syntax Error

The first category E1 describes syntactical errors. The additional operation `inherit` of `Kernel::Class` (page 87 in [6]) is a typical example for this category.

```
inherit(inhs:Set(Kernel_NamedElement)):Set(Kernel_NamedElement)=
  inhs->excluding( inh | ownedMember
    ->select( oclIsKindOf(Kernel_RedefinableElement) )
      ->select( redefinedElement->includes(inh) ) )
```

The defining expression of the operation `excluding` is syntactically incorrect, because the bar `|` is not allowed in this place, as the original OCL syntax shows [7]:

```
set->excluding(object:T):Set(T)
```

As an example for a parameterized error sub-category for syntax errors we mention 'E1.1(boolValue)' which describes all errors, where a boolean value was used like an enumeration value. An example occurs in the following invariant of `IntermediateActions::AddStructuralFeatureValueAction` (page 219 of [6]):

```
inv: let insertAtPins=self.insertAt in
     if self.structuralFeature.isOrdered=#false
       then insertAtPins.isUndefined()
       else ... endif                        -- details left out
```

The literals `true` and `false` are predefined values in OCL, which means that the sharp `#` has to be omitted. The following further sub-categories of syntax errors E1 were identified as well:

- E1.2(X): 'X' is missing in the expression. With 'X' element of {(,),[,],...}.
- E1.3: Parentheses '[' ']' are not allowed here.
- E1.4: Too many right parentheses.
- E1.5: 'endif' is missing in the expression.
- E1.6: 'if' expression must have an 'else' expression.

2.2 Category E2: Minor Inconsistency

Errors which result from minor inconsistencies are described in the second category E2. One representative example for this category provides the following operation `is` of `Kernel::MultiplicityElement` (page 240 of [6]):

```
is(lowerbound:Integer, upperbound:UnlimitedNatural):Boolean=
  (lowerbound=self.lowerbound and upperbound=self.upperbound);
```

The identifiers (probably intended attributes or rolenames) `self.lowerbound` and `self.upperbound` are not defined in the context class or one of its superclasses. However there are derived attributes `lower` and `upper` and additional operations `lowerbound()` and `upperbound()` declared in this context. Probably one of these both possibilities is meant.

The single error sub-category identified within category E2 represents errors resulting from an erroneous multiplicity check:

- E2.1(X): Use OCL construct 'X.isDefined()' instead of 'X.multiplicity.is(1,1)' which is not defined as an operation (X being a rolename).

The navigation 'multiplicity' is not allowed in connection with 'X'.

2.3 Category E3: Type checking error

Very common OCL traps are type checking errors which are denoted here by E3. Type checking errors are easy to make and difficult to recognize. The operation `visibleMembers` is defined within the class `Kernel::Package` (page 100 of [6]) and shows the difficulty of this error category:

```
visibleMembers():Set(Kernel_PackageableElement)=
  member->select( m | self.makesVisible(m))
```

As Fig. 1 shows, the identifier `member` refers to a rolename of an association between the superclasses `Kernel::Namespace` and `Kernel::NamedElement`. Thus the above operation body returns a set of `Kernel::NamedElement`. This type is not compatible with the return type `Set(Kernel::PackageableElement)`.

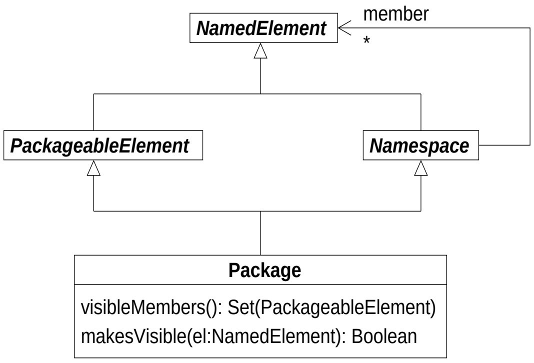

Fig. 1. Context Class Diagram for Operation `visibleMembers()`

One more example of this error category can be found in Appendix A.

2.4 Category E4: General Problems

Category E4 characterizes rather general problems. One typical example deals with the following additional operation of `BehaviorStateMachines::StateMachine` (page 491 of [6]):

```
LCA(s1:BehaviorStateMachines_State,s2:BehaviorStateMachines_State)
  :BehaviorStateMachines_State=
if ancestor(s1,s2) then s1 else
  if ancestor(s2,s1) then s2 else
    LCA(s1.container,s2.container) endif endif
```

The operation `LCA` possesses two parameters of type `BehaviorState-Machines::State`. However the recursive call of `LCA` assigns objects of type `BehaviorStateMachines::Region` to these parameters. This assignment leads to a type mismatch.

In Appendix B one further example of this category can be found.

'E4.1(member)' is a sub-category within E4 and describes errors where an attribute or rolename that is identified by the parameter 'member' is not defined. As an example consider the first invariant of `Kernel::Association` (page 81 of [6]) which uses the identifier `memberEnd`.

```
inv: self.parents()->forAll( p |
  p.memberEnd.size()=self.memberEnd.size() )
```

An attribute or rolename `memberEnd` is not defined in the UML 2.0 superstructure specification in this context. We also mention the last sub-category of the general problems which will not be described in detail:

- E4.2(X): Class 'X' is not defined (X being a class).

For us it was interesting that the OCL syntax check revealed places where the UML 2.0 class diagrams were inconsistent, i.e., places where names of attributes, rolenames, or classes are used but these names are not properly declared. Therefore, the OCL check offers much more than the analysis of the pure OCL expressions: The OCL check also makes statements about the correctness of the UML 2.0 class diagrams.

2.5 Category E5: Inconsistency with USE

Errors that utilize unsupported features of USE are described in the final category E5. One representative example for this category can be found in the following invariant of the class `Templates:: RedefinableTemplate-Signature` (page 557 of [6]):

```
inv: inheritedParameter = if extendedSignature->isEmpty()
       then Set{}
       else extendedSignature.parameter
     endif
```

The construct `Set{}` is not supported by USE. Instead USE offers the syntax `oclEmpty(Set(X))` where the specific type `X` must be substituted. The type of these expressions, e.g., `oclEmpty(Set(State))` or `oclEmpty(Set(String))`, is then uniquely defined in contrast to `Set{}` which has many types. In this case `Set{}` must be replaced by `oclEmpty(Set(Templates::TemplateParameter))`.

2.6 Changes Made before Validating

Before we were able to employ USE, we had to make the following small modifications to the original description:

- We appended an underscore to identifiers which are keywords in USE, e.g., instead of the rolename `class` we use `_class`.
- The respective package names were added to the class names, e.g., instead of `Kernel::Namespace` we employ `Kernel_Namespace`, because USE is unaware of packages.
- The new primitive data type `UnlimitedNatural` was realized as a class, because it is not supported by USE.

All other necessary changes were explicitly identified in the validation process as error category E5 'Inconsistency with USE' and can be viewed in detail in the provided EXCEL file [1].

3 Results of the Validation

This section deals with the achievements of the validation. In the following, the structure of the EXCEL file and statistical issues will be discussed.

3.1 Error Documentation Structure in the EXCEL file

Every recognized error is documented in the mentioned EXCEL file [1]. The errors can be identified by their page location in the UML 2.0 Superstructure Specification [6]. The columns of the EXCEL file can be classified as follows:

- Page number of the offending entity (if applicable),
- page number of the corresponding diagram (if applicable),
- package name of the enclosing package,
- nature of the offending entity (e.g., class or invariant),
- context classifier of the offending entity,
- name of the participating classes (for associations, if applicable),
- type of the offending member (e.g., operation or rolename),
- name of the offending member,
- our error category,
- our error description, and
- location in our related USE file [1].

Figure 2 shows an example line from the EXCEL file (rotated by 90 degrees) which corresponds to the above example of the E4 category. This format is also the format in which a directly readable version of the EXCEL file is provided, namely the PDF file in [1].

Column	Column description
Page	page 491 (Acrobat Reader page 507)
Figure	Figure 354, page 457 (Acrobat Reader page 473)
PackageName	BehaviorStateMachines
Classifier	Class
ClassEnumName	StateMachine
ParticipatingClassNamesForAssocs	(not applicable in this example)
MemberKind	operation
MemberRoleInvName	LCA(s1:BehaviorStateMachines::State, s2:BehaviorStateMachines::State)
ErrorCategory	E4
ErrorDescription	Operation call of LCA has 's1.container' as parameter. 's1.container' is of type 'Region' but a 'State' is expected.
USELine	1443

Fig. 2. Example Line of the EXCEL File

3.2 Error Statistics

The UML 2.0 Superstructure contains 36 packages with 294 classes and 14 enumerations. 246 invariants and additional operations were defined with OCL and were taken into account during the validation process. We discovered 361 errors within these 246 OCL expressions. But also inconveniences and incompatabilites of our tool USE are included within this number.

Packages	#OCL exprs	#E	ERatio	#E1	#E2	#E3	#E4	#E5
AssociationClasses	2	7	3.5	2			5	
BasicActivities	1	2	2.0	1			1	
BasicInteractions	2	3	1.5				3	
BehaviorStateMachines	38	46	1.2	18		6	22	
Communications	1	1	1.0				1	
CompleteActions	26	27	1.0	6	9		12	
Fragments	1	2	2.0				2	
Interfaces	1	0	0.0					
IntermediateActions	51	105	2.0	28	15	5	51	6
Kernel	98	104	1.0	24	1	24	31	24
ProtocolStateMachines	2	0	0.0					
StructuredActivities	2	8	4.0	5			3	
Templates	17	20	1.1	3		4	8	5
UseCases	4	4	1.0	1				3
Associations		32					13	19
Sums	246	361		88	25	39	152	57
Percentages		100%		24%	7%	11%	42%	16%

Fig. 3. Numbers for the Validation

The table in Fig. 3 lists all packages of UML 2.0 that use OCL. There are also 22 packages which do not use OCL at all. The line 'Associations' does not refer to a package, but to the errors which occurred during association handling, e.g., due to wrong rolenames. The table gives the number of OCL expressions (invariant or operation definition) in that package and shows the number of errors encountered. The five columns #E1, #E2, #E3, #E4, and #E5 in the right of the table refine the total number of errors #E according to our error categories. The fourth column entitled ERatio gives the number of errors per OCL expression. Accordingly, if one regards only our numbers, there are good and bad packages, e.g., the package StructuredActivities has the highest ERatio (4.0), the packages Interfaces and ProtocolStateMachines have the lowest ERatio (0.0). But both packages use only few OCL expressions. We found it also interesting that even packages with intensive use of OCL, namely the packages having more than ten OCL expressions, have a relatively high ERatio, i.e., an ERatio between 1.0 and 2.0.

4 Conclusion

In this paper, we have analyzed the OCL expressions in the UML 2.0 superstructure. Within the 246 OCL expressions we have found 361 places where our OCL tool USE had complaints. Many errors had its origin not in bad use of OCL, but in deficiencies of the underlying class diagrams. We think that all parts of an important standard like UML should be error-free and have provided detailed information about the errors and possible corrections in the accompanying EXCEL file.

However, in our validation we have not exhausted the power of our validation tool USE. USE allows for example to instantiate the class diagrams by snapshots. With such an instantiation one could show the consistency of the invariants (Are there contradictory invariants?). Furthermore, in principle USE also allows to check whether one invariant is a consequence of a given set of invariants. Thus a redundancy check is imaginable as well (Are all invariants independent from each other?). Nevertheless, in addition to checks performed by machines one will still need developers enhancing the design quality through model inspection and discussions made with other modeling experts.

Acknowledgements

We would like to thank Bran Selic for pointing to the importance of the OCL part within UML 2.0.

References

1. Hanna Bauerdick, Martin Gogolla and Fabian Gutsche. UML 2.0 Validation Results in Form of an EXCEL, PDF, AND USE File. University of Bremen. 2004. ftp://ftp.informatik.uni-bremen.de/local/db/papers/uml2004/ocl_uml2.[xls|pdf|use].
2. IBM Corporation, Rational Software: Rational Rose. http://www.rational.com.
3. Mark Richters and Martin Gogolla. Validating UML Models and OCL Constraints. In Andy Evans, Stuart Kent and Bran Selic, editors, UML 2000 - The Unified Modeling Language. Advancing the Standard. Proc. 3rd International Conference, LNCS 1939, pages 265-277. Springer, 2000.
4. Jim Rumbaugh, Ivar Jacobson and Grady Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, 2nd Edition, 2004.
5. OMG. Unified Modeling Language Specification, Version 1.5, March 2003. Object Management Group, Inc., Needham, MA, http://www.omg.org, 2003.
6. OMG. UML 2.0 Superstructure Specification. Version ptc/03-08-02. Object Management Group, Inc., Needham, MA, http://www.omg.org, 2003.
7. Jos Warmer and Anneke Kleppe. The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, 2nd Edition, 2003.

A Further Example for Error Category E3

One further interesting error example can be found in the second invariant of class `Templates::NamedElement` (page 560 of [6]).

```
inv: nameExpression.oclIsKindOf(Kernel_LiteralString)
       implies name = nameExpression.value
```

First of all there exists an inconsistency between the superstructure pdf document from the OMG and the superstructure MDL file. The rolename `nameExpression` references in the OMG document an association between `Templates::NamedElement` and `Templates::Expression`. However, in MDL file this rolename leads us to the class `Templates::StringExpression`.

Independent from the above inconsistency, this invariant will always be evaluated to true since the expression `nameExpression` can never be of type `Kernel::LiteralString` as you can conclude from the class diagram in Fig. 4.

B Further Example for Error Category E4

The following excerpt from an invariant of class `IntermediateActions::-LinkEndCreationData` (page 238 of [6]) reveals another interesting error which ocurrs quite often within the superstructure document.

```
... insertAtPin.type = UnlimitedNatural ...
```

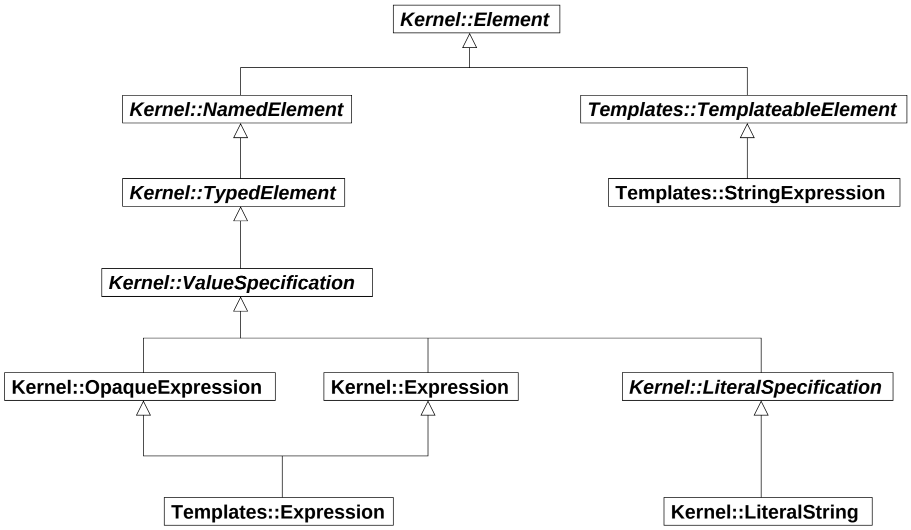

Fig. 4. Context Class Diagram for second invariant of class `Templates::NamedElement`

The rolename `type` returns an instance of `Kernel::Type`. With the use of the equality `=` operator the types of `insertAtPin.type` and `UnlimitedNatural` are tried to be compared. However in OCL the type comparison must be realized with the operation `oclIsTypeOf`.