Integrating TwoUse and OCL- DL

Studienarbeit

in computer science

presented by

David Saile

Advisor: Fernando Silva Parreiras, FB4

Koblenz, january 2010

Contents

Chapter 1
Introduction

Abstract This section introduces the context of this work. We mention the problem, and the motivation that led to the start of this research. Finally a short overview over the scope of this work and the remaining chapters is given.

1.1 Context

OCL [1] is a modeling language to create and enhance software models. In general, OCL is used in combination with modeling languages like the *Unified Modeling Language (UML)* [2] because every OCL expression is based on a type, defined in a diagram. Formally designed only for UML, OCL may now be used with any Meta-Object Facility (MOF [3]) Object Management Group (OMG [4]) metamodel, including UML. Information like specific constraints and behavioral specifications for methods of UML classes is not expressible by UML. The task of OCL is to add information to object-oriented models, that is not expressible by the diagrams themselves. The information added to a model by OCL is expressed in application-specific *constraints*. Further, OCL can be used to specify completely programming language independent expressions on models, that can be evaluated by an interpreter over a specific model instance.

1.2 Problem

In recent development, attempts have been made to integrate UML and OWL into one hybrid modeling language, namely TwoUse [5]. This aims at making use of the benefits of both modeling languages and overcoming the restrictions of each. In order to create a modeling language that will actually be used in software development an integration with OCL is needed. This integration has already been described at the contextual level in [6], however an implementation is lacking so far.

1.3 Motivation

The integration of TwoUse and OCL allows the software developer to make use of the predefined OCL standard library with several predefined operations. Since TwoUse introduces the integration of ontologies into UML modeling, it is obvious to also make use of the benefits provided by the ability of ontologies to query a reasoner. A reasoner can infer logical consequences from a model, and therefore return information, not explicitly modeled in the diagram. This leads to the scope of this work to not only integrate OCL and TwoUse on the programing level, but to also extend the OCL standard library with several predefined operations, that make use of ontology reasoning.

1.4 Scope of this work

[6] introduces the integration of TwoUse with OCL. The scope of this paper is the programatical implementation of the integration of TwoUse with OCL. In order to achieve this, two different OCL implementations that already provide parsing and interpretation functionalities for expressions over regular UML, as well as an implementation of the conventional OCL standard library are introduced. The attempt of an implementation of our goal, that extends the first OCL implementation produced problems, that could easier be dealt with using the second OCL implementation. This paper presents the two attempts to extend the OCL implementations, as well as a comparison of both approaches.

1.5 Overview

The remainder of this paper is organized as follows: Chapter 2 presents the background for this work, and introduces the most important concepts. Based on these we tried to extend two different OCL implementations, that each implement OCL including its standard library. Our aproaches to extend them are presented in chapter 3 and 4 respectively. Chapter 5 compares the two approaches based on our experiences, before chapter 6 finally concludes.

Chapter 2
Background

2.1 Introduction

In this section the context for this *Studienarbeit* is presented. The context of the results, presented here is a new modeling approach called *TwoUse*. Since the goal of this *Studienarbeit* is to integrate the *Object Constraint Language* (OCL) and *TwoUse* on the programatical level, both of these concepts are introduced. First there will be a short description of the use of OCL in *Model-driven development*. Second the concepts and ideas of *TwoUse* are explained. Since there are several papers about both of these topics, there is no need to go much into detail.

2.2 The example model

To show the new possibilities created by TwoUse and to explain several operations and problems on an example, we chose an existing example from the literature, that has already been used in the original TwoUse paper [6]. We use the development of an international e-commerce system that is described it in a model as an example. The corresponding class diagram can be seen in Figure 2.1.

This example has been elaborated in [7]. The system represents a sales order system for Canada and the United States. The taxes and total amount a customer has to pay is calculated depending on the country the products are delivered to. The taxes can be divided into Government Sales Tax (GST) and Provincial Sales Tax (PST) for sales going to Canada. Sales orders are controlled by the class TaskCtrl. The SalesOrder itself is either a USSalesOrder or a CanSalesOrder, according to the Country the Customer lives in. The operation freight() on which we will focus later on queries the country of the customer and returns the corresponding freight amount, depending on whether the order is a USSalesOrder for a citizen of the United States or a CanSalesOrder for a citizen of Canada. Since UML class diagrams alone are not expressive enough to demonstrate the behavior of this oper-

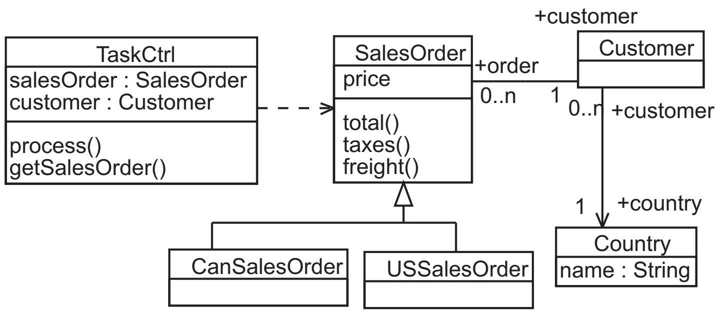


Fig. 2.1 SalesOrder

ation, a textual query language such as OCL [1] may be used to specify such a query:

```
context SalesOrder::freight(): Integer
  body:
    if self.owlIsInstanceOf(CanSalesOrder)
        then 10
    else
        if self.owlIsInstanceOf(UsSalesOrder)
            then 5
        else
            20
        endif
    endif
```

2.3 The Object Constraint Language OCL

2.3.1 Overview

OCL is a declarative language to specify expressions over models, that cannot be expressed in a diagram. Initially it was only used in combination with UML [2], but due to its high popularity it was extended to be used with any MOF OMG metamodel. The OCL standard defined by the OMG [8] distinguishes the following purposes for which OCL can be used:

- As a query language
- To specify invariants on classes and types in the class model
- To specify type invariant for stereotypes
- To describe pre- and post conditions on operations and methods
- To describe guards
- To specify target (sets) for messages and actions
- To specify constraints on operations

- To specify derivation rules for attributes for any expression over a UML model.

For explanations of the different items, please see the OCL specification [8].

2.3.2 Basic elements in OCL

In OCL, every value has a specific type no matter if it is an object, a class or an instance of a component or a datatype. OCL types are grouped into

- predefined types (as defined in the OCL standard library), including
  - basic types
  - collection types
- custom types (defined by users)

Integer, *Real*, *String* and *Boolean* are predefined basic types, comparable with datatypes in other languages. *Collection*, *Set*, *Bag*, *OrderedSet* and *Sequence* are the predefined collection types, used to specify concrete results of a navigation over compounds of a class diagram. Custom types like the classes Customer and SalesOrder in our example are defined by users in an UML-diagram. Every model element of an UML-diagram that can be instantiated is automatically an OCL type. A special type that has to be mentioned specifically is OclAny, since it's the supertype of all the types, except for the pre-defined collection types. OclAny itself is an instance of the metatype AnyType. All classes in an UML model inherit all operations defined on OclAny (you can recognize them on the prefix 'ocl' (e.g. *oclIsKindOf()*).

2.3.3 Relation to the corresponding model

Each OCL expression is written in the context of an instance of a specific type. This instance can be referred to by the variable *self* and is called *classifier*. Since there already exist some predefined operations in OCL, the type of the context specifies which operations are applicable. For example if the OCL expression is written in the context of an instance of the type *SalesOrder* (see the example model in Fig. 2.1), only the operations corresponding to *SalesOrder* an its supertypes are applicable. In this case we could use the operation *freight()* because it is defined in the model, but also operations that are defined in the supertype of *SalesOrder*, namely *AnyType* (e.g. *oclIsKindOf()*).

2.3.4 Operations

Operations are bound to a specific OCL type, and are applicable only in the context of this type. This means, that they can only be applied to instances of this type, and to instances of subtypes of this type. Some operations are defined on the previously mentioned OCL super type *OclAny*. This makes these operations available for all types. ”In general, OCL allows the definition of additional operations and attributes using **def:** expressions. This is very convenient for the formulation of constraints” [9] but for the definition of complex operations that are supposed to become a part of the OCL standard library another mechanism is used. The OCL standard library is a collection of predefined types (as described in section 2.3.2) and operations.

2.3.5 OCL in eclipse

2.3.5.1 EMF OCL

Many different programatical approaches to OCL have been made in the past. It is not intended to compare these here. Since we used the Eclipse software platform in our project from the beginning, we initially used the eclipse OCL project [10]. The eclipse OCL project provides the basic library *org.eclipse.ocl* that provides a definition of the extensible environment API for OCL parsing and evaluation. It consists of the basic interfaces that have to be implemented by a concrete metamodel to create and evaluate OCL expressions. *org.eclipse.ocl.ecore* and *org.eclipse.ocl.uml* are implementations of an OCL binding for a concrete metamodel. Here, the concrete mechanisms to parse and evaluate OCL constraints on models are implemented. Further, these packages ”extend the types of the OCL Types package to define the generalization relationships to the ... metamodel’s counterparts to the UML Classifier (EClassifier) and DataType (EDataType) metaclasses. This ensures a consistent type system in the OCL binding for Ecore, so that all types are represented as EClassifiers” [11]. An implementation of an example OCL interpreter (*org.eclipse.emf.ocl.examples.interpreter*) already exists. Therefore, we would not have to implement one ourselves from scratch but would be able to just extend this one, since it is open source. Another nice fact is, that the TwoUse metamodel is created under the Ecore metamodel. Since the OCL interpreter already allows to evaluate OCL expressions on Ecore models, we should be able to extend this implementation (package *org.eclipse.ocl.ecore*).

2.3.5.2 Dresden OCL Toolkit

The Dresden OCL Toolkit is an other implementation of OCL for eclipse. It consists of a collection of libraries, allowing for integrated use of modeling languages like UML with OCL. The toolkit is provided in several different versions, one of which is a plug-in collection for Eclipse. The core of the toolkit is a pivot model. This model works as an abstraction layer between the metamodel of the modeling language and the metamodel of the OCL library. This allows for evaluation of OCL

expressions over instances of arbitrary domain-specific languages (DSL) without adapting the OCL implementation to the DSL.
Beside the adaption of metamodels to OCL, the Dresden OCL Toolkit enables the user to parse and interpret OCL expressions as well as generating Java code, enforcing constraints specified in OCL.

2.4 TwoUse

2.4.1 Motivation

In *Model-Driven Engineering* different modeling approaches exist, with different strengths and weaknesses. Therefore they become appropriate for the specification of different aspects of software systems. Two of these modeling approaches are *UML* [2] and *OWL* [12]. The idea of TwoUse is to combine these approaches in a coherent framework for developing integrated models, comprising the benefits of UML models and OWL ontologies and overcoming their restrictions. While mappings from one model to the other have already been established a while ago, the goal of TwoUse is to be able to denote them in a hybrid diagram.

2.4.2 Problem

Considering the example class diagram of an e-commerce system in section 2.2, we see that some operations need to be modeled with additional information, presented for example in OCL. Unfortunately the declarations of the classes USSalesOrder and CanSalesOrder occur at least twice: ”once in the class declaration and once implicitly, as an expression of the operation TaskCtrl.freight()” [6]. We try to avoid such redundancy in the context of TwoUse by describing the specific type of SalesOrder exactly once. For this we use the language OWL, capable of logical class definition which is in fact more expressive than UML. The OWL diagram, corresponding to our example domain would look like depicted in figure 2.2.

2.4.3 Idea

The ideal solution is a model using the advantages of both UML and OWL models, namely a TwoUse model. TwoUse was developed in order to fulfill six basic requirements:

1. *Full Expressiveness of UML an OWL.*
2. *Cross-referencing* from UML part to the OWL part and vice versa.

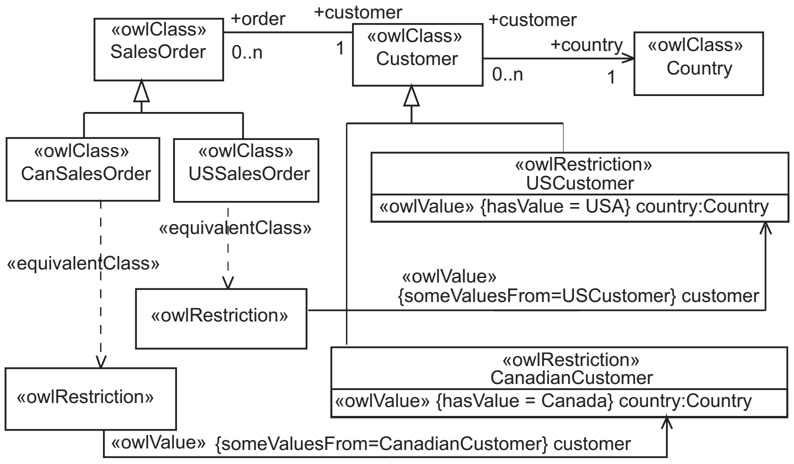

«owlClass» SalesOrder
+order
0..n
+customer
1
«owlClass» Customer
+customer
+country
0..n
1
«owlClass» Country
«owlClass» CanSalesOrder
«owlClass» USSalesOrder
«owlRestriction» USCustomer
«owlValue» {hasValue = USA} country:Country
«equivalentClass»
«equivalentClass»
«owlRestriction»
«owlValue» {someValuesFrom=USCustomer} customer
«owlRestriction» CanadianCustomer
«owlValue» {hasValue = Canada} country:Country
«owlRestriction»
«owlValue» {someValuesFrom=CanadianCustomer} customer

Fig. 2.2 Ontology of the SalesOrder example.

3. *Compatibility with UML Tools.*
4. *Concrete Syntax for Hybrid Diagrams*
5. *Metamodeling Support*
6. *Model-driven Engineering*

Two use is based on four core ideas to fulfill these requirements, that were originally presented in [6]. First, a MOF based metamodel is provided that integrates UML, OWL, and OCL. Second, an *UML profile* is used to provide a syntactic basis. This choice was made in order to support standard UML2 extension mechanisms. It also enables mappings from the profile onto TwoUse models. The third idea is to provide a canonical set of transformation rules to the user. This aims at enabling the integration at the semantic level. The fourth idea is the most important one from the point of view of this paper. In order to make use of the reasoning capabilities of OWL ontologies, an extension of the OCL basic library called OCL-DL is introduced.

The TwoUse metamodel (fig. 2.3) imports the OWL, UML and OCL metamodel.

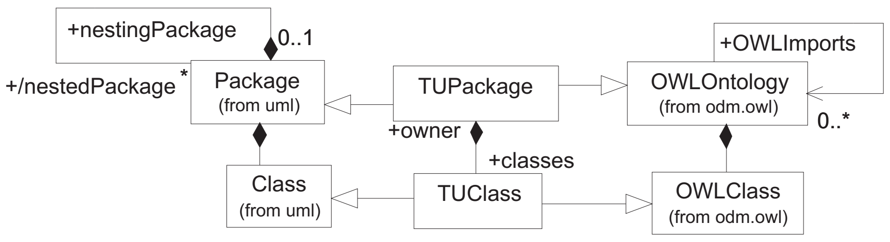


Fig. 2.3 TwoUse Metamodel

A big advantage of the integration of OWL is the ability to use a reasoner. Reasoners can be used to check the consistency of a model, which means that it makes sure that there are no contradictions. It can further be used to check for concept satisfiability. This makes sure, that a concept definition can actually be fulfilled, for example whether a class can have an instance. Third, reasoners can be used to classificate concepts. This allows to conclude information about the relationship of different concepts to each other. An example here is the question, whether every instance is automatically an instance of a certain class. Last, a reasoner can classificate an instance, which means it can determine whether it is an instance of a certain class, or not. To use these features provided by the possibility to use the OWL reasoner, an extension to OCL called *OCL-DL* is introduced.

2.4.4 OCL-DL

For the best use of TwoUse, existing and additional OCL expressions are needed. So our goal was not only to implement a metamodel binding for TwoUse, but also to provide additional, TwoUse-specific operations. Since time did not permit the actual implementation, instructions where and how to implement the operations are given later. [6] already introduced OCL-DL, an extention of the OCL standard library with pre-defined operations that call the OWL reasoner. Our goal is the actual implementation.

2.4.4.1 Problem

To demonstrate the use of OCL-DL we consider our running example, described in section 2.2. Let us consider the case of a class CanSalesOrder, which is a subtype of the class SalesOrder. To be more specific we state, that the SalesOrder class must have an attribute Customer. The object in this attribute must itself have an attribute Country with the value 'Canada' to be a CanSalesOrder. There are two predefined OCL operations for the type *OclAny*, namely

1. oclIsTypeOf(typespec: **OclType**): **Boolean**
2. oclIsKindOf(typespec: **OclType**): **Boolean**

The first one, *oclIsTypeOf* returns true if the model element it is applied to is of the type of its argument. So applied to an instance of the class CanSalesOrder, the operation would only evaluate to true, if its argument is the class CanSalesOrder. The second operation *oclIsKindOf* returns true if the model element it is applied to is of the type of its argument or is a subtype of it. Applied to an instance of the class CanSalesOrder, the operation would evaluate to true, if its argument is the class CanSalesOrder or it's supertype SalesOrder, or any supertype of SalesOrder (e.g. OclAny). The problem here is the scenario, if we have an instance of the class SalesOrder and ask if it is also an instance of the class CanSalesOrder (i.e. its attribute Customer has the value ’Canada’ in its attribute Country). In this context, both of the operations described above would evaluate to false. ”The reason is that, in any object-oriented paradigm, the compiler cannot dynamically subtype classes based on their descriptions” [6].

2.4.4.2 Solution

Since TwoUse-classes can be hybrids of UML and OWL-classes, the getSalesOrder() operation can be specified without using complex OCL expressions, but by taking advantage of the OWL part of TwoUse by querying an *OWL reasoning service*. To decide whether a class is a subclass of an other class we introduce an OCL-DL named *owlIsInstanceOf* that makes use of a reasoner and tests if a given instance is an instance of the operation’s argument by checking if it meets the given constraints.

```
context TaskCtrl::getSalesOrder():SalesOrder
  body:
    salesOrder.oclAsType(
                              salesOrder.owlMostSpecNamedClass()
    )
```

In the context of this operation an OCL-like query may check whether the given SalesOrder instance fulfills all the logical requirements of USSalesOrder or CanSalesOrder. The basic OCL operation oclAsType does the casting in the subclass, referred by its parameter. To determine which subclass of SalesOrder is applicable, the OCL-DL operation owlMostSpecNamedClass is invoked. This operation ”queries a reasoner to return the reference to the suitable named subclass according to the OWL ontology.” [6] The advantage of this kind of specification of getSalesOrder() is the separation of two sources of specification complexity. The specification of complex classes remains only in the OWL model and the specification of the operations needed by the developed system remain only in the OCL expression, and stay small, more comprehendible and easily maintainable by the use of the OWL class definition and the application of the reasoner. The same is the case for the SalesOrder operation freight which calculates the freight amount of an order depending on the type of the SalesOrder. As presented in the listing in section 2.2, freight can be specified using OCL including the OCL-DL operation *owlIsInstanceOf*, without querying the home country of a coustomer to determine the kind of SalesOrder it is applied to. This avoids the redefenition of the SalesOrder subclasses in the body of freight.

[6] introduces four OCL-DL operations namely:

1. owlIsInstanceOf(typespec: **OclType**): **Boolean**. Evaluates to **true** if the object satisfy all the logical requirements of the OWL class description **typespec**.
2. owlAllNamedClass(): **Set(OclType)**. Returns all named classes classified by a reasoner, whose the object satisfies the logical requirements.

3. owlAllInstances():**Set(T)**. This is an introspective operation which returns all instances that satisfy the logical requirements of the OWL class description of the given object.
4. owlMostSpecNamedClass():**Ocltype**. Returns the intersection of owlAllNamedClass().

[6]

Chapter 3
First approach: Extending the EMF OCL implementation

3.1 Introduction

In this section, the existing projects we initially based our research on are presented. While working on this approach several problems occurred, that are described in section 3.7 and led to the decision to choose a different approach, presented in chapter 4.

3.2 The OCL interpreter

The OCL interpreter can be found in the package *org.eclipse.emf.ocl.examples.interpreter*. ”This example-interpreter illustrates the usage of the generic OCL Parser API to parse and evaluate OCL query expressions and constraints within the SDK.” [13] It extends the Ecore example editor with a console, that provides two fields. You can enter an OCL expressions in the bottom field and press **Enter** to evaluate them on the model, while output and errors are shown in the top field.

The console provides the opportunity to choose (using a *Drop-Down-Action*) whether to evaluate expressions either on an UML or an Ecore model. ”These actions automatically select the appropriate metamodel in the console.” [13] Further, the user is able to determine the model level on which the query is ’executed’, namely either the model-level *M1* or the metamodel-level *M2*. While parsing of expressions is provided on both the M1 and the M2 level, the evaluation of expressions is only implemented for the M2 level.

3.3 Adding custom operations

The first task was to determine how to add custom operations. In the EMF OCL implementation, the basic steps are to create a custom *Environment* that knows how to look up this operation, and an *EvaluationEnvironment* that knows how it is implemented. Concrete, the *Environment* has to extend either the *AbstractEnvironment* (this means you have to do a lot of work on your own, but sometimes is inescapable) or an existing Environment implementation (e.g. the *EcoreEnvironment*). The new Environment needs a constructor and a mechanism to define the new operations and add them to a type. A possible solution can be seen in listing 3.1.

Listing 3.1 ”An Environment to look up the custom operation *owlIsInstanceOf*”

```
class MyEnvironment extends EcoreEnvironment {
  EOperation regexMatch;

  // Initialize the root environment
  MyEnvironment(EPackage.Registry registry) {

    super(registry);
    defineCustomOperations();
  }

  ...

  // Add our custom operation to OCLAny
  private void defineCustomOperations() {

    owlIsInstanceOf = EcoreFactory.eINSTANCE.
        createEOperation();
    owlIsInstanceOf.setName("owlIsInstanceOf");
    owlIsInstanceOf.setEType(getOCLStandardLibrary().
    getAnyType());

    // Create and add its parameter
    EParameter parm = EcoreFactory.eINSTANCE.
    createEParameter();
    parm.setName("pattern");
    parm.setEType(getOCLStandardLibrary().
    getAnyType());
    owlIsInstanceOf.getEParameters().add(parm);

    // Annotate it so that we will recognize it
    // in the evaluation environment
    EAnnotation annotation =
 EcoreFactory.eINSTANCE.createEAnnotation();
```

```
    annotation.setSource("MyEnvironment");
    owlIsInstanceOf.getEAnnotations().add(annotation);

    // define it as an additional operation on
    // OCL AnyType
    addOperation(getOCLStandardLibrary().getAnyType(),
    owlIsInstanceOf);

  }
}
```

Next the corresponding *EvaluationEnvironment* needs to be implemented. Again, we extend the *EcoreEvaluationEnvironment* so it knows how to handle calls to the added custom operation:

Listing 3.2 ”An EvaluationEnvironment to execute the code for *owlIsInstanceOf*”

```
class MyEvaluationEnvironment extends
                EcoreEvaluationEnvironment {

  MyEvaluationEnvironment() {

    super();
  }

  MyEvaluationEnvironment(
        EvaluationEnvironment<EClassifier,
          EOperation, EStructuralFeature,
          EClass, EObject> parent) {

    super(parent);
  }

  public Object callOperation(EOperation operation,
        int opcode, Object source, Object[] args) {

    if (operation.getEAnnotation("MyEnvironment")
                                  == null) {
      // not our custom owlIsInstanceOf operation
      return super.callOperation(operation, opcode,
                                  source, args);

    }else if ("owlIsInstanceOf".equals(
                        operation.getName())) {
     //Here belongs the implementation of the operation

    }else{
        // unknown operation
        throw new UnsupportedOperationException();

    }
  }
}
```

The final step is to create an *EnvironmentFactory* that creates the custom environments.

3.4 Evaluation on the M1 level

As mentioned in section 3.2, the original OCL console did only allow for evaluation on the M2 level. Some of the **OCL-DL** operations we want to define operate on the M1 level. For example the operation *owlIsInstanceOf* is applied to an instance, gets a class as its argument, and queries, whether all constraints that need to be fulfilled for an object to be an instance of this class can be evaluated to true. Therefore we needed to enable evaluation on the M1 level. The first step was to alter the *OCLConsolePage* class, specifically in the case distinction of the modeling level at case *M1*, not only parsing, but also evaluation of the query had to be enabled. Using the the same code as in case *M2* produced wrong results. So we had to make code changes on the concrete implementation of each operation (e.g. *oclIsTypeOf*, *oclIsKindOf*, ...). Since they are implemented differently and at different locations for different model elements, we had to work on the implementation of every single one.

3.5 Adding custom operations to a model and making them available at runtime

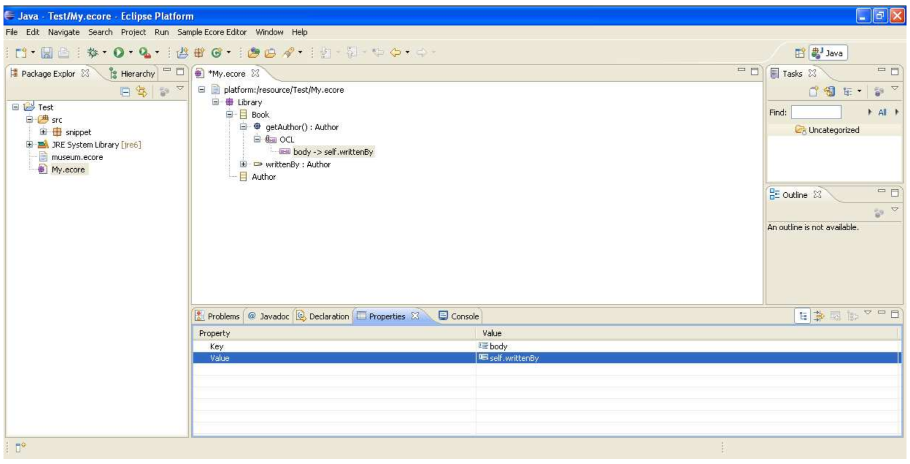
3.5.1 Adding OCL expression to model elements

The next idea was to allow the user to create operations in the model, and enhance them with OCL expressions that describe their behavior. The user should be able to execute these operations on instances of the model he created. The *EMF sample editor* already allows the user to add *EOperations* to ecore models (and TwoUse models, since ecore is the metamodel of TwoUse). The editor also provides the opportunity to add *EAnnotations* to a model element. These can further be extended with *Details Entries* that consist of a *Key* and a *Value*. We decided to use the following mechanism:

1. The user adds an *EOperation* to the metamodel.

2. This operation is annotated with the *EAnnotation* **”OCL”**.
3. The *EAnnotation* gets a *Detail Entry* with the *key* **”body”**. The OCL expression describing the operation is written to the *value* field.

To enhance user-defined operations with OCL expressions we chose this mechanism, because it sounded very convenient and intuitive to us. Of course the other advantage was that we could use existing mechanisms. Another possibility could have been to add a field called *body* to the properties-view of the *ecore sample editor* and let the user write his OCL expression there. The problem was the fact that instead of being able to use the existing editor, we would have had to extend the *EMF sample editor*, and also add the modified project as one of our plug-ins instead of just using the version provided by the *EMF* plug-in. Since from the beginning our goal was to edit as few existing plug-ins as possible, we chose to use the *EAnnotation* mechanism as shown in figure 3.1.


Java - Test/My.ecore - Eclipse Platform
File Edit Navigate Search Project Run Sample Ecore Editor Window Help
Package Explor
Hierarchy
*My.ecore
Tasks
Find:
All
Uncategorized
Outline
An outline is not available.
Problems
Javadoc
Declaration
Properties
Console
Java

```
Test
  src
    snippet
  JRE System Library [jre6]
  museum.ecore
  My.ecore
```

```
platform:/resource/Test/My.ecore
  Library
    Book
      getAuthor() : Author
        OCL
          body -> self.writtenBy
      writtenBy : Author
    Author
```

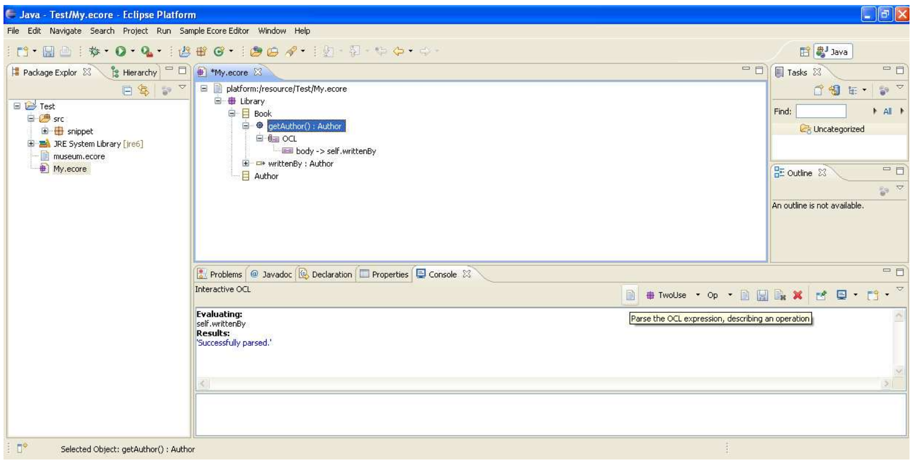
Property	Value
Key	body
Value	self.writtenBy

Fig. 3.1 EAnnotation

It is important, that the OCL expression describes the operation adequately so it can be fully evaluated at runtime. To check whether a written expression is semantically correct, the user has to be able to parse his OCL expression.

3.5.2 Parsing

Again, the first idea was to add this function to the existing editor, specifically to the window into which the user writes the OCL expression. Because of the same reasons as in the previous section, and also the dependencies this solution would create we chose a different approach. Since this task is performed by the console, the most convenient solution to us was to add a button to it, that allows the user to parse the OCL expression of the currently chosen operation (as shown in figure 3.2).


Java - Test/My.ecore - Eclipse Platform
File Edit Navigate Search Project Run Sample Ecore Editor Window Help
Package Explor
Hierarchy
*My.ecore
Tasks
Find:
All
Uncategorized
Outline
Problems
Javadoc
Declaration
Properties
Console
TwoUse
Op
Parse the OCL expression, describing an operation

```
Test
  src
    snippet
  JRE System Library [jre6]
  museum.ecore
  My.ecore
```

```
platform:/resource/Test/My.ecore
  Library
    Book
      getAuthor() : Author
        OCL
          body -> self.writtenBy
      writtenBy : Author
    Author
```

An outline is not available.

Interactive OCL

```
Evaluating:
self.writtenBy
Results:
'Successfully parsed.'
```

Selected Object: getAuthor() : Author

Fig. 3.2 Parsed custom operation

We then only had to ensure that the parsing of both, M1 and M2 expressions would succeed. This is done, by first parsing the expression as an M1 expression. The code for this parsing (listing 3.3 lines 14 - 22) is surrounded by a try-catch block, that catches any occurring exception (most likely a *ParserException*).

Listing 3.3 ”Code to parse custom operations”

```
1  else if(((EOperation)context).getEAnnotation("OCL").
       getDetails().get("body") == null)

     error(OCLInterpreterMessages.console_emptyOclExpr);

5
   else {
     String expression = ((EOperation)context).
       getEAnnotation("OCL").getDetails().get("body");

10 ocl = oclFactory.createOCL(ModelingLevel.M1);
   OCLHelper<Object, ?, ?, ?> helper = ocl.
     createOCLHelper();

   boolean m1parse = false;
15 try {
     ...

     m1parse = true;
     OCLEvaluationModelingLevel.modelingLevel = 1;
```

```
  helper.createQuery(expression);
  print(OCLInterpreterMessages.console_parsed,
        outputResults,
        false);

} catch (Exception e1) {
if(m1parse){
  // M1 parsing already began => maybe the exception
  // is a ParsingException, that occurred because the
  // expression is no M1, but a M2 expression.
  // Therefore we try to parse it as a M2 expression.

  print(
    OCLInterpreterMessages.except_during_M1_parsing,
    colorManager.getColor(ColorManager.OUTPUT_RESULTS),
    false);

  ocl = oclFactory.createOCL(ModelingLevel.M2);
  OCLHelper<Object, ?, ?, ?> helperM2 =
    ocl.createOCLHelper();

  try{
    ConstraintKind kind = ModelingLevel.M2.setContext(
        helperM2, context, oclFactory);
      ...

    helperM2.createQuery(expression);
    print(OCLInterpreterMessages.console_parsed,
          outputResults,
          false);

  } catch (Exception e2){
    // Both M1 and M2 parsing failed
    // => expression is no valid OCL
    error((e2.getLocalizedMessage() == null)
           ? e2.getClass().getName()
           : e2.getLocalizedMessage());
  }
} else {
  // M1 parsing didn’t begin yet, so the Exception
  // was thrown before. Therefore there is no use in
  // trying to parse it as an M2 expression.
  error((e1.getLocalizedMessage() == null)
         ? e1.getClass().getName()
         : e1.getLocalizedMessage());
}
}
```

If the exception occurred during the parsing or evaluating of the OCL expression (shown by the m1parse flag) it is very likely that this exception occurred, because it is actually a M2 expression. Therefore the catch-block handles the exception by restarting the parsing from the beginning, but this time parsing the expression as a M2 expression. Again this part is surrounded by try-catch block. If an exception occurs again, the expression is not valid OCL.

3.5.3 Enable evaluation at runtime

The next idea was to let the user execute the operations he added, on the instances of his model. Luckily the operations provided in the metamodel are available in the *OCL console*, when the metamodel they are described in is registered. This functionality (registering a model as metamodel) is already provided by the AM3 plug-in [14]. The custom functions are even available on the content-assist of the console (ctrl+space). So our task was to implement the evaluation of the operation. For this we used the *EvaluationEnvironment* we created in the process of adding *OCL-DL* operations to the **OCL standard library**. As shown in listing 3.3 (lines 1 - 8) we only had to check, if the chosen operation has an *EAnnotation* with the value **”‘OCL”’** and ensure that the *DetailEntry* with the key **”body”** is not null. Once this was ensured we could use the code of the *OCLConsolePage*’s evaluation method to evaluate the body as a query.

3.6 Adding a metamodel binding for TwoUse

3.6.1 Abstract approach

Before the actual implementation of the new operations, a metamodel binding for TwoUse has to be established (the TwoUse metamodel can be found in fig. 2.3). While determining how to add custom operations, we noted that the metamodel binding does not occur in the *org.eclipse.emf.ocl.examples.interpreter* plug-in itself, but in the org.eclipse.ocl.ecore respectively org.eclipse.ocl.uml package. This interfered with our original plan to just extend the interpreter plug-in. So we had to create a package called *org.eclipse.ocl.twouse* that realizes the metamodel binding for TwoUse and will be used by the interpreter. To create a metamodel binding ”you need to ensure that your metamodel implements the constructs required by OCL in some fashion: Operation, Property, etc. Your metamodel must be Ecore-based (Ecore is the meta-metamodel).” [15]

3.6.2 Classes to implement

If these requirements are satisfied, you need to create a metamodel specific plug-in, that serves as the connection between the metamodel and OCL. This plug-in (*org.eclipse.emf.ocl.twouse*) has to implement Classes for the following concepts:

- EnvironmentFactory
- Environment
- UMLReflection
- OCLStandardLibrary
- EvaluationEnvironment

An *EnvironmentFactory* implementation is needed, since it creates the *Environment* and the *UMLReflection* classes. The implementation of the *Environment* requires suitable substitutions for the generic type parameters representing the metamodeling constructs required by OCL to provide relations between the OCL metamodel and your metamodel. The *UMLReflection* is needed for introspecting models (instances of the target metamodel), while the implementation of the *OCLStandardLibrary* as an instance of your metamodel provides the instances of the metamodel's Classifier metaclass that implement the OCL standard library types. An *EvaluationEnvironment* is needed that knows how your metamodel 'works' for accessing properties of run-time instances of models. These are the basic premises to create a metamodel binding for OCL. Our approach is presented in 3.6.3.

3.6.3 Concrete

As described in section 3.6.1, the main entry point to create a binding for a custom metamodel is the implementation of an *EnvironmentFactory*, with generics that suit your metamodel. Our approach was to create, a separate plug-in for TwoUse similar to *org.eclipse.ocl.ecore*, named *org.eclipse.ocl.twouse*.

3.6.3.1 Renaming generics - the idea

The first step was to create generics belonging to our metatmodel (*TUClass*, *TUClassifier*, *TUOperation*, ...). Since in the beginning, our package should behave exactly like the ecore package for simplicity reasons, the first idea was to copy all the ecore generics (*EClass*, *EClassifier*, *EOperation*, ...) and just rename them from *E...* to *TU...* (e.g. *EClass* → *TUClass*). This is possible, because eclipse projects are open-source, and their source code can be downloaded from the CVS repository. The exact 'tuning' of the interfaces belonging to the generics to adjust them to the TwoUse metamodel was supposed to follow later. Since the generics were now behaving exactly like the ecore ones, it was convenient to just copy the the *EcoreEnvironmentFactory*.

3.6.3.2 Renaming generics - the problem

Unfortunately this caused a lot of errors, since now methods from other packages, called in method bodies in our package were not called with the correct types any more. Some of these errors could be solved by adding casts, but some of them required code changes in the called method itself. One problem that occurred were methods with `EList<EClass>` as return-type. After changing the return-type to `EList<TUClass>`, the called method itself had to be altered, so it returns an *EList* of the new type, but again the same error occurred, that methods called within this method returned ecore types and sometimes, again a cast was not enough. This lead to some deep and very complex changes in code, that soon crossed the boarder to other packages, imported by *org.eclipse.ocl.ecore* (e.g. *org.eclipse.emf.ecore*). Since we wanted to alter as few packages as possible, and the code-changes became very complex and numerous, we had to think of a new approach.

3.6.3.3 Creating interfaces that extend the ecore generics

The next idea was to create interfaces, that simply extend the corresponding ecore generics (EClass, EClassifier, EOperation, ...) and instantiate the TUEnvironment-Factory with them.

```
import org.eclipse.emf.ecore.impl.EClass;

public interface TUClass extends EClass{
}
```

Again, we copied and renamed the *EcoreEnvironmentFactory* but now, since we had different argument-types and return-types for the implemented methods, their signatures had to be changed (EClass $\rightarrow$ TUClass). This raised another problem: since some methods called methods that had the 'old' ecore generics as parameter or return-type, again there were many errors in the new *TUEnvironmentFactory*. In conclusion, we came to the same problem as in the first approach and discarded this idea as well.

3.6.3.4 Copy the ecore plug-in

Since the implementations of the model elements (*EClassImpl*, *EClassifierImpl*, *EOperationImpl*, ...) are located in *org.eclipse.emf.ecore*, this plug-in had to be copied as well, to allow code changes to be made here. This allowed us to create a closed workspace (see figure 3.3), that contained the console plug-in (*org.eclipse.emf.-ocl.examples.interpreter*), the plug-in providing the metamodel binding for TwoUse (*org.eclipse.ocl.twouse*) and the plug-in containing the model implementations (*org.-eclipse.emf.ecore*). The only other plug-in needed by the interpreter is *org.eclipse.ocl*.

Since this plug-in provides the basic OCL concepts, there was no need to make any changes there, and it could be added to the build-path as a *jar*.

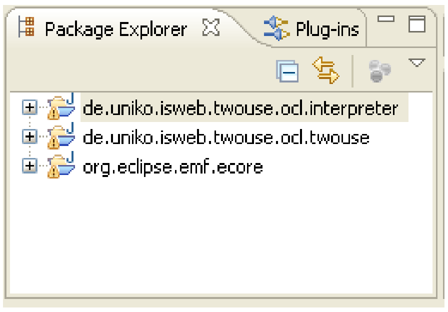

Package Explorer
Plug-ins
de.uniko.isweb.twouse.ocl.interpreter
de.uniko.isweb.twouse.ocl.twouse
org.eclipse.emf.ecore

Fig. 3.3 Closed workspace

3.7 Problems

Because of the problems we experienced while creating a binding for the TwoUse metamodel we came to the conclusion, that there is no use in trying to extend or modify the *org.eclipse.ocl.ecore* plug-in. A complete programatical implementation of the TwoUse metamodel would be needed, that needs to be written from scratch. Another approach, that addresses this problem in a very convenient way is the Dresden OCL toolkit for Eclipse. It solves the process of creating a metamodel binding for a custom DSL in a far more elegant way. This led us to the decision to discard the first approach using the EMF-OCL implementation, and use the Dresden OCL Toolkit as the platform to implement our OCL-DL operations.

Chapter 4
The second approach: Extending the Dresden OCL toolkit

4.1 Introduction

In this section, the second approach of extending an existing OCL toolkit is presented. This involves adding a metamodel binding for TwoUse and modifying the OCL parser in order to be able to parse the OCL-DL operations. It also comprises implementing the method bodies of these operations to be able to interpret expressions using these new standard library operations, as well as extending the java code generation from OCL constraints, provided by the Dresden OCL toolkit. This chapter presents our approach in dealing with these tasks.

4.2 Design

The Dresden OCL toolkit [16] is implemented as an eclipse plug-in, and provides an implementation of OCL including the OCL standard library. The toolkit enables users to parse and interpret OCL expressions over model instances, as well as generating java code for the specified expressions. The plug-in consists of the following parts:

- Pivot model
- OCL standard library implementation
- OCL parser
- OCL interpreter
- Java code generator

The most notable aspect of the toolkit is the pivot model, which is kind of a proxy that is situated between the OCL metamodel and the metamodel of the target models. This provides an intermediate abstraction layer, to allow the user to connect any custom metamodel to the toolkit. Instantiation of the OCL standard library types becomes independent of the used metamodel, and therefore the only necessary steps are the suitable implementation of the pivot model interfaces.

4.3 Adding metamodel binding

4.3.1 Problem binding the TwoUse metamodel

4.3.1.1 Idea

Adding a metamodel binding, allowing to use the OCL implementation with a custom DSL is very easy using the Dresden OCL toolkit. Since the EMF approach caused us a lot of problems creating the metamodel binding, this was the main motivation to choose this OCL implementation. In the Dresden OCL toolkit, binding the metamodel only requires establishing bindings from the custom DSL classes to the corresponding pivot classes [17].

4.3.1.2 Problem

The difficult part here is the fact, that the TwoUse metamodel contains two types, i.e. *UMLClass* and TwoUse Class (*TUClass*), where the second one is a subclass of the first one, but also inherits from OWL ontologies (as described in [6]). The problem is, that both of these classes must have a corresponding OCL metaclass, allowing for the newly implemented special OCL-DL operations to be applied to instances of *TUClasses*, but not to instances of regular *UMLClasses*. This stems from the fact, that the OCL-DL operations make use of an ontology reasoner that requires the queried model instance to inherit from OWL. The problem is, that the existing OCL metamodel only has one supertype for all types, namely *OclAny*. As described in section 2.3, all operations owned by *OclAny* (e.g. *oclIsKindOf*) are applicable to all instances that are subtypes of *OclAny*, i.e. all types. As can be seen in figure 4.1, the type *UMLClass* is connected to the pivot class *Type*, which is on the other hand connected to the OCL type *AnyType*. To make the distinction between *UMLClass* and *TUClass* while parsing the OCL expressions, a mechanism to decide whether an object is an instance of a *TUClass*, and therefore applicable to OCL-DL operations is needed in the OCL engine.

4.3.2 OwlAny

Our idea was to add a subclass to *OclAny*, called *OwlAny*, as well as adding a subclass to *Pivot:Type* called *TuType* and finally connecting *TuType* with *OwlAny* on one side, and *TUClass* on the other (as depicted in figure 4.1). After trying to modify the existing Dresden OCL toolkit implementation in order to add these new classes we experienced a lot of errors that were difficult to resolve. Therefore we contacted the developers of the toolkit who were very doubtful regarding the possibility to make these changes without rewriting huge parts of the toolkit. Since the only need for this modification is the ability to throw parsing errors when OCL-DL operations are applied to UML classes, we decided to move this error detection to the code generation and interpretation process, and leave the task to implement a suitable parsing error detection for this case for the future.

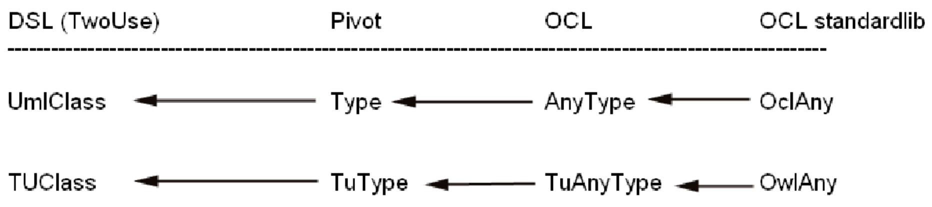


Fig. 4.1 Idea how to bind the TwoUse metamodel to the pivot model

4.4 Adding OCL-DL operations

For dresden an approach is described in [18], that is similarly simple as the adding of a metamodel, described in the previous section. This approach relies on simply modeling the new operations in a model of the standard library, namely *oclstandardlibrary.types*, which can be found in the package *tudresden.ocl20.pivot.modelbus*. After adding a new operation to the model, a rebuild of the whole project is supposed to add the necessary code to parse and interpret the new operation. Unfortunately this is not fully implemented yet, so in praxis methods to parse and interpret a new operation have to be added to several classes of the toolkit.

4.4.1 Concrete implementation of owlIsInstanceOf

This section describes how to exemplarily implement one of the OCL-DL operations. Due to the fact that a metamodel binding for TwoUse is still missing, and therefore the bridge from UML class specifications to the OWL reasoner cannot be established yet, the scope of this work does not allow to include a complete implementation of one of the OCL-DL operations. However, we enabled parsing for *owlIsInstanceOf*, and the following describes where exactly the respective changes have to be made in order to add and implement the OCL-DL operations. This should serve as kind of a manual for future implementation of the operations. The implementations of the operations differ only very little. In fact, there are only very few places where the code for the other operations will be different than the code for *owlIsInstanceOf*. Mainly, this will be the actual implementation of the respective operation (discussed in section 4.4.1.2) and the template code for the java code generation from OCL expressions (discussed in section 4.4.1.3).

4.4.1.1 Parsing

The main entry point is the '.types' model of the OCL standard library. It can be found in the *tudresden.ocl20.pivot.modelbus.resources* folder and is called *oclstandardlibrary.types*. In order to enable parsing for an operation that is supposed to be added to the standard library, the operation needs to be modeled here. As described in the beginning of this section, it is not implemented yet to automatically generate the rest of the code from here, but mechanisms to parse the operation have to be added to several java classes.
First of all, the operations signature has to be added to the *tudresden.ocl20.pivot.-sessentialocl.standardlibrary.OclRoot* interface. This was done by by creating another interface called *OclRootTu* in the same package, that extends the *OclRoot* interface and implements the operation's signature. Next, several files in the *tudresden.ocl20.pivot.ocl2parser.gen.parserfiles* package have to be modified in order to enable parsing. This can be done accordingly to other, already implemented operations (e.g. *oclIsKindOf*). Finally the actual parser file *ocl2.parser* in the *tudresden.ocl20.pivot.ocl2parser* plug-in is automatically generated by running the ant-script [19].

4.4.1.2 Interpreting

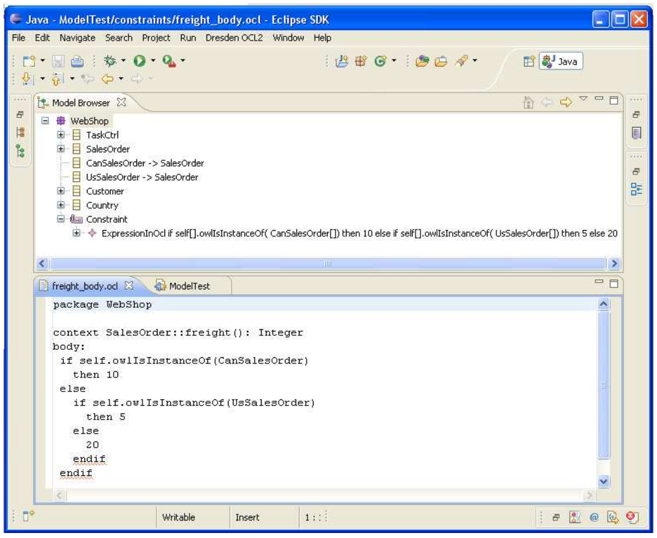
The actual implementation of the operations themselves has to be done in *tudresden.ocl20.pivot.standardlibrary.java.internal.library.JavaOclRoot*. As mentioned previously, what keeps us from implementing the operations, is a missing connection between the (UML) Class and the ontology. This connection is needed, in order to allow a reasoner to infer knowledge about a class, which is the main idea behind the OCL-DL operations [6]. A possible approach has been discussed in section 4.3.2, however a working implementation is lacking so far.

4.4.1.3 Code generation

To make use of the ability of the *Dresden OCL Toolkit* to generate *AspectJ code*, the templates which are responsible for generating the code have to be extended. The template that needs to specify what kind of code needs to be generated for a certain operation is *tudresden.ocl20.pivot.ocl2java.resources.template.java.operations.stg*. Here an entry for the respective operation that is supposed to be added has to be included, along with the code specifying the AspectJ code that will be generated.

4.4.2 Testing

In order to test the implementation of *owlIsInstanceOf* the example model that has been introduced in section 2.2 can been used. We created an Ecore model of the *WebShop* and, generated model code from this model. This can simply be done by using the EMF functionality to generate java code from a *genmodel*. After creating the model code, an OCL query using the newly added OCL-DL operation *owlIsInstanceOf* has been created as described in section 2.2 (the OCL query can be seen in section 2.2). Since we added the signature of *owlIsInstanceOf* to the standard library, the query is now successfully parsed (as can be seen in figure 4.2).


Java - ModelTest/constraints/freight_body.ocl - Eclipse SDK
File Edit Navigate Search Project Run Dresden OCL2 Window Help
Java
Model Browser
freight_body.ocl
ModelTest
Writable
Insert
1 : :

```
WebShop
  TaskCtrl
  SalesOrder
  CanSalesOrder -> SalesOrder
  UsSalesOrder -> SalesOrder
  Customer
  Country
  Constraint
    ExpressionInOcl if self[].owlIsInstanceOf( CanSalesOrder[]) then 10 else if self[].owlIsInstanceOf( UsSalesOrder[]) then 5 else 20
```

```
package WebShop

context SalesOrder::freight(): Integer
body:
 if self.owlIsInstanceOf(CanSalesOrder)
   then 10
 else
   if self.owlIsInstanceOf(UsSalesOrder)
     then 5
   else
     20
   endif
 endif
```

Fig. 4.2 A parsed OCL impression that uses the newly added owlIsInstanceOf operation

We used the Dresden OCL toolkit functionality to generate AspectJ code (as described in [20]) to create code from the parsed OCL constraint. This example can be used together with the *main-method* shown in listing 4.2 to test the implementation of *owlIsInstanceOf*, once it is implemented. So far, it only shows that the AspectJ template code for *owlIsInstanceOf* has been added in the right places.

Listing 4.1 ”The generated AspectJ code for the OCL body of *freight*”

```
package WebShop.constraints;

@Generated
public privileged aspect BodyAspect1 {

    /**
     * <p>Pointcut for all calls on
     * {@link WebShop.SalesOrder#freight()}.</p>
     */
    protected pointcut freightCaller(
   WebShop.SalesOrder aClass):
     call(* WebShop.SalesOrder.freight())
     && target(aClass);

    /**
     * <p>Defines the body of the method freight()
     * defined by the constraint
     * <code>context SalesOrder::freight(): Integer
     *       body: if self[].owlIsInstanceOf(
     *    CanSalesOrder[])
     *            then 10
     *             else if self[].owlIsInstanceOf(
     *         UsSalesOrder[])
     *                  then 5
     *                  else 20</code></p>
     */
    Integer around(WebShop.SalesOrder aClass):
   freightCaller(aClass) {
        Integer ifExpResult2;

        if (aClass.owlIsInstanceOf(
   WebShop.CanSalesOrder)) {
            ifExpResult2 = new Integer(10);
        } else {
            Integer ifExpResult1;

            if (aClass.owlIsInstanceOf(
     sWebShop.UsSalesOrder)) {
                ifExpResult1 = new Integer(5);
            } else {
                ifExpResult1 = new Integer(20);
            }
            ifExpResult2 = ifExpResult1;
        }

        return ifExpResult2;
    }
}
```

Listing 4.2 ”The *main-method* to test the generated AspectJ code for *owlIsInstanceOf*”

```
package WebShop.testing;
```

```java
import WebShop.Country;
import WebShop.Customer;
import WebShop.SalesOrder;
import WebShop.impl.CanSalesOrderImpl;
import WebShop.impl.CountryImpl;
import WebShop.impl.CustomerImpl;

public class WebShopTest {


 public static void main(String[] args) {
  Country canada = new CountryImpl();
  canada.setName("Canada");
  Country usa = new CountryImpl();
  usa.setName("USA");

  Customer david = new CustomerImpl();
  david.setHomeCountry(canada);

  SalesOrder ord1 = new CanSalesOrderImpl();
  ord1.setOwner(david);
  ord1.setPrice(50);

  float fr = 0;
  fr = ord1.freight();
  assert(fr == 10);
  float total = ord1.getPrice() + fr;

  System.out.println("Total amount is ",total);
 }

}
```

Chapter 5
Comparison of EMF and Dresden OCL toolkit

5.1 Introduction

The EMF implementation of OCL with the example interpreter (*emf*) and the Dresden OCL toolkit (*dresden*) are two useful, but diverse approaches of implementing OCL for eclipse and providing means to parse and interpret expressions. This section aims at comparing these approaches, since we used both of them in trying to implement OCL-DL. Several aspects that distinguish the two approaches are described in the following.

5.2 Metamodel binding

The first aspect that was important to us, is the way the two approaches enable the binding of a customized metamodel. As described in section 3.6.1, *emf* requires the user to implement several classes in a way that is specific to the metamodel. This requires deep understanding of the architecture of the *emf* plug-ins and is quite complicated. *Dresden* solved this in a far more elegant approach. As described in [17], the user can bind his metamodel by only mapping the elements of his metamodel, to the corresponding pivot model elements. This does not require any programming, and can be done by simply following the provided tutorial with only very little metamodeling knowledge.

5.3 Adding operations to the OCL standard library

Since the main goal of this work was to add new operations to the OCL standard library, the way to do this was similarly important as adding a metamodel binding. For *dresden*, several changes have to be made to various classes, as explained in

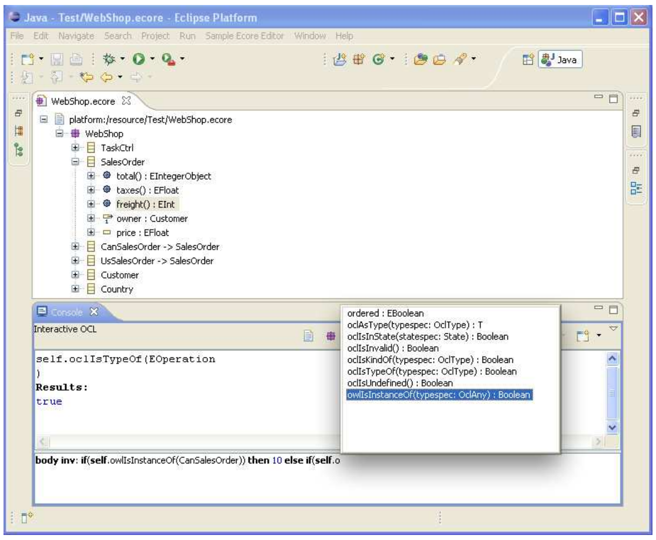
section 4.4. This cannot be done in a clear and cohesive way. The developer must explore which classes and packages to alter, since no actual description of how to do this could be found. Emf on the other hand also requires code changes to be made, but a clear description of the classes that need to be modified, and a description of the changes that need to be performed in order to add new operations can be found in [9]. This has already been elaborated in section 3.3.
So in theory, *dresden* offers a more elegant approach as described in 4.4, which in praxis has not been fully implemented yet. The mechanism provided by *emf* on the other hand is well documented and easy to implement.

5.4 Addressing questions

Another aspect that led us to the decision to choose the *dresden* implementation is the way questions and problems can be addressed. The Dresden OCL toolkit has been developed at the *Technische Universität Dresden*, therefore a team of developers can directly be contacted. Further, an OCL Discussion list exists under `Dresden-ocl-discussion@lists.sourceforge.net`. For *emf* on the other hand an eclipse newsgroup exists [15], where problems can be discussed. Unfortunately there is very little participation in this newsgroup.

5.5 The plug-ins

The different layouts and ways to use the plug-ins are worth to be discussed as well. In *dresden* the model, and an instance of the model if needed, can be loaded. Each of them is opened in its own tab in the eclipse view (see figure 4.2). *Xmi*-files containing OCL expressions and constraints can then be loaded and parsed. The successfully parsed constraints are displayed in the model instance tab. The user can then choose a specific constraint, and evaluate it over the model instance. The results are displayed in an additional tab.
Emf offers a different approach. The model and model instances are opened as separate windows in the EMF model editor. The OCL interpreter console is opened as a view tab, providing a section for entering expressions and a section displaying the results of an evaluated expression (see figure 5.1). Like the *dresden* plug-in the console allows the import of xmi-files containing OCL expressions. Further, direct input of expressions over the input field is possible. By pressing *return*, the user parses and evaluates the query over a selected model object, and the results are displayed in the upper section of the console.


Java - Test/WebShop.ecore - Eclipse Platform
File Edit Navigate Search Project Run Sample Ecore Editor Window Help
Java
WebShop.ecore
Console
Interactive OCL

```
platform:/resource/Test/WebShop.ecore
  WebShop
    TaskCtrl
    SalesOrder
      total() : EIntegerObject
      taxes() : EFloat
      freight() : EInt
      owner : Customer
      price : EFloat
    CanSalesOrder -> SalesOrder
    UsSalesOrder -> SalesOrder
    Customer
    Country
```

```
self.oclIsTypeOf(EOperation
)
Results:
true
```

```
ordered : EBoolean
oclAsType(typespec: OclType) : T
oclIsInState(statespec: State) : Boolean
oclIsInvalid() : Boolean
oclIsKindOf(typespec: OclType) : Boolean
oclIsTypeOf(typespec: OclType) : Boolean
oclIsUndefined() : Boolean
owlIsInstanceOf(typespec: OclAny) : Boolean
```

```
body inv: if(self.owlIsInstanceOf(CanSalesOrder)) then 10 else if(self.o
```

Fig. 5.1 The EMF OCL console to parse and interpret OCL expression

Chapter 6
Conclusion

6.1 Solved Problems - Added functionalities

6.1.1 Problems

In order to extend the OCL standard library we had to overcome a number of problems. One that posed itself during the research for this work, was the decision which existing implementation should be extended. In fact two toolkits existed that were suitable candidates for our work, i.e. the EMF OCL implementation with the example interpreter, and the Dresden OCL toolkit. This paper presented both approaches, the work we did trying to extend them, their differences, and the reasons that led us to chose the Dresden implementation. The most important aspect in our decision, and also the task that presented the most problems was the adding of a metamodel binding for the TwoUse metamodel. Unfortunately we were only able to solve this task partly, as discussed in the previous chapters.

6.1.2 Added functionalities for EMF

For the EMF example interpreter we enabled M1 model level interpreting (in addition to the existing M2 model level interpreting). We also added the functionality to enhance the ecore-model-element *EOperation* with OCL definitions that implement the behavior of this operation. These OCL definitions can be parsed and evaluated, which enables the user to use these operations in OCL expressions he enters into the console. Further we added the signature of one of the OCL-DL operations to the OCL standard library (i.e. *owlIsInstanceOf*), enabling parsing but not interpreting expressions using this operation. Interpretation could not be implemented since we had problems adding a metamodel binding for TwoUse for this approach.

6.1.3 Added functionalities for dresden

For the Dresden OCL toolkit we also added the the signature of the operation *owlIsInstanceOf* to the OCL standard library as an example of one of the OCL-DL operations. This allows for parsing OCL expressions making use of this operation as one of the new standard library functions. Unfortunately we were not able to fully implement interpretation and code generation for this function. This is due to the problems we had in finding a connection between the TwoUse metamodel and the OCL metamodel. However we gave instructions how and where to add future implementations for the OCL-DL operations, in order to enable interpreting as well as code genration, once these problems have been overcome.

6.2 Future Tasks

Because of several problems that occurred during this work that have been elaborated in previous sections, we were unable to add a proper metamodel binding for TwoUse. Fortunately this was tolerable in our final implementation that extends the Dresden OCL toolkit. TwoUse Models can simply be imported as UML models, leaving TwoUse classes as simple UML classes. The disadvantage here is, that OCL-DL operations can be added to, and parsed with simple UML classes that do not have reasoning ability. An error wont be thrown before interpretation of the OCL query or the execution of generated aspectj code. As a future task, a concrete metamodel binding for TwoUse should be added, that allows the differentiation between UML and TwoUse classes as well as error detection of OCL-DL operations that are applied to UML classes.
As a result of the numerous problems we were not able to implement one of the OCL-DL operations. While this is left as a future task, this paper offers instructions on how exactly to to this.

6.3 Conclusion

In the scope of this work we explored two approaches that implement the OCL standard library, compared both of them and tried to enable the use of OCL with TwoUse models. This posed several problems that could be partly overcome by extending the Dresden OCL toolkit. We also tried to add one of the OCL-DL operations introduced in [6], enabling users to create OCL expressions for TwoUse models, that use this operation. The advantage of these operations is, that they make use of the reasoning ability of TwoUse models. OCL expressions containing the added OCL-DL operation can be parsed, but not interpreted and used to generate AspectJ code yet. In order to enable users to make use of the complete OCL-DL library, some more work needs to be done in the future.

References

1. Group, O.M.: Object Constraint Language Specification, version 2.0. Object Modeling Group. (June 2005)
2. Group, O.M.: Uml resource page. `http://www.uml.org/`
3. Group, O.M.: Official mof specification from omg. `http://www.omg.org/spec/MOF/2.0/`
4. Group, O.M.: Official website of the object management group. `http://www.omg.org/`
5. Koblenz-Landau, U.: Transforming and weaving ontologies and uml in software engineering (twouse). `http://www.uni-koblenz.de/FB4/Institutes/IFI/AGStaab/Projects/twouse`
6. Parreiras, F., Staab, S., Winter, A.: Using ontologies with uml-based modeling: The twouse approach. (2007)
7. Shalloway, A., Trott, J.R.: Design Patterns Explained: A New Perspective on Object-Oriented Design (2nd Edition) (Software Patterns Series). Addison-Wesley Professional (2004)
8. Group, O.M.: OCL specification
9. Foundation, T.E.: Ocl developer guide → programmer's guide → advanced topics → customizing the environment. `http://help.eclipse.org/ganymede/index.jsp`
10. Foundation, T.E.: The eclipse foundation. the object constraint language (ocl) project. `http://www.eclipse.org/modeling/mdt/?project=ocl` (2007)
11. Foundation, T.E.: Ocl developer guide → reference → ocl api reference → org.eclipse.ocl.ecore. `http://help.eclipse.org/ganymede/index.jsp`
12. Consortium, W.W.W.: OWL Web Ontology Language Reference
13. Foundation, T.E.: Ocl developer guide → examples guide→ ocl interpreter example. `http://help.eclipse.org/ganymede/index.jsp`
14. Foundation, T.E.: The eclipse foundation. atlas megamodel management (am3) project. `http://www.eclipse.org/gmt/am3/` (2007)
15. Foundation, T.E.: Subject: 'other model backends'. `http://www.eclipse.org/newsportal/thread.php?group=eclipse.modeling.mdt.ocl`
16. Dresden, T.U.: Project pages of the dresden ocl toolkit. `http://dresden-ocl.sourceforge.net/`
17. Dresden, T.U.: Pivot model adapter generation. `http://dresden-ocl.sourceforge.net/papers/pivotAdapterGen.pdf`
18. Bräuer, M., Demuth, B.: Model-level integration of the ocl standard library using a pivot model with generics support. (2008) 182–193
19. Dresden, T.U.: Readme. tudresden.ocl20.pivot.ocl2parser.README.txt
20. Dresden, T.U.: How to use the java code generator ocl2java. `http://dresden-ocl.sourceforge.net/4eclipse_usage.html`