Entwicklung eines OCL-Parsers für UML-Extensionen

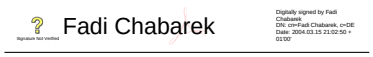
Diplomarbeit

Betreut durch:
Prof. Dr. H. Weber
Technische Universität Berlin

02.03.2004

Fadi Chabarek

Die selbständige und eigenhändige Anfertigung versichere ich an Eides statt.
Berlin, den 02.03.2004



Unterschrift

Development of an OCL-Parser for UML-Extensions

Thesis

Fadi Chabarek
Technical University of Berlin

02.03.2004

Abstract

In this thesis a validator for UML Extensions is developed and implemented. Therefore an interpreter framework for OCL 1.5 constraints is build. The framework chains a parser, a context checker and an interpreter. These components access required model information using a model interface that describes arbitrary models in terms of OCL concepts. To support MOF 1.4 compliant metamodels, this interface is implemented using JMI. Based on MDR, interpreter implementations for the MOF Model and the UML metamodel in the version 1.3 and 1.4 are realized. The latter implementation can be used to validate UML Profile extensions.

Contents

Chapter 1

Introduction

In this time of globalization of information and accompanying growth in the heterogeneity of data the exchange of metadata becomes an increasingly relevant subject. Metadata is data about data. When used to describe structural and semantical properties for modeling purposes, metadata expresses a model on a higher level of abstraction: a metamodel.

The Unified Modeling Language (UML) [UML03] defines one of the most widespread metamodels. It can be adapted to individual styles of modeling to suite systems or architectures and bridge heterogeneity in software engineering processes. This is achieved through an extension mechanism employing so called profiles [UML03, cha. 2.6]. Profiles narrow properties of the metamodel down to domain specific needs by supporting the definition and application of constraints for individual metamodeling elements. These additional constraints on the metamodel are usually expressed in the Object Constraint Language (OCL) [OCL03]. OCL mixes features of object-oriented (OO) languages like navigation and namespaces with first order logic. It was originally conceived by IBM for financial projects using OO-architectures. Those projects had to avoid the ambiguities of natural language and provide domain experts with a simplified equivalent of mathematical formalisms. Similar problems arose during the formalization of the UML and OCL was subsequently included in the standard. A UML model is thus defined as a model that meets the requirement of the static class-based structure of the metamodel and whose elements satisfy all OCL constraints defined on these classes. For the latter part, these constraints need to be evaluated. Stereotypical

elements which make up domain specific profiles carry additional constraints and also have to be evaluated. Although the UML has been in use for a long time, such evaluators are not available, leaving correctness of models a matter of guesswork and hand-waving. This thesis provides such an evaluator along with an extensible validation framework.

1.1 Validation Framework Architecture

For the purpose of validating OCL constraints on the varying versions of the UML metamodel and its instances, a validation framework is designed. Every version of the UML metamodel is an isolated model instance described by a metamodel itself. This metametamodel is the Meta Object Facility (MOF) [MOF02] Model. It forms the core of the validation framework.

The validator follows the classical pipe-and-filter architecture [Pep97] chaining a parser, context checker and interpreter (s. figure 1.1).

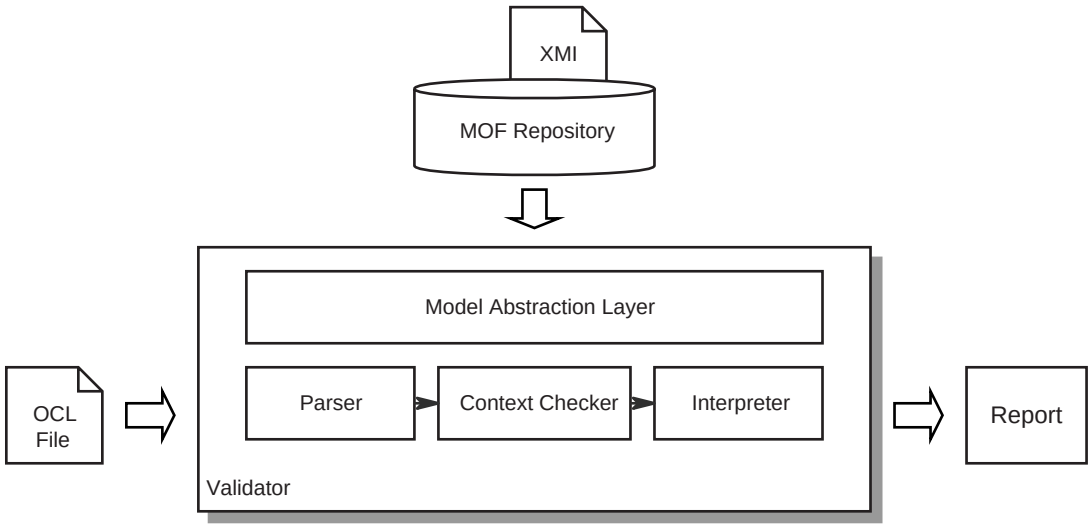


Figure 1.1: The Validator Architecture

The input consists of a set of constraints and a description of a model instance and its metamodel. In order to make this description amenable to analysis, a model

abstraction layer is developed. It consists of an abstract model and a bridge to the MOF. On the one hand the abstract model acts as an interface controlling the access to and the information about OCL concepts included in the model input. On the other hand the MOF Bridge supports MOF compliant metamodels as it implements the abstract model with the corresponding model descriptions. The import of a MOF compliant metamodel described by the XML Metadata Interchange (XMI) [XMI03] format into a concrete MOF repository technology enables our framework to interpret and validate OCL constraints on corresponding model instances. This especially allows us to validate constraints defined on the UML metamodel in its varying versions and subsequently respective UML Profiles.

1.2 Thesis Structure

Chapter 2 analyzes and adjusts OCL with respect to inconsistencies and inaccuracies. It builds the foundation the validator is based on. Chapter 3 discusses the relation between arbitrary models and OCL. It develops the model interface based on the type and instance level to import a model into the OCL context. The interpreter chain is presented in chapter 4 through chapter 6. Chapter 4 adjusts the OCL grammar to be LALR(1)-parseable and develops a corresponding parser. Chapter 5 presents the context checker. The OCL type system is partitioned into predefined and model types. OCL predefined types are mapped to Java types so the Java Reflection API [Mic97] provides access to the properties of the OCL type structure. The integration of model types obtained from the model interface completes the design of the type system. Context dependent OCL features and types are checked. Chapter 6 reuses these types to develop the interpreter. Analogously to the type check an evaluation algorithm is defined. To represent and access validation results of constraints the model interface is extended. Chapter 7 discusses the relation between MOF metamodels and OCL. Consistently the MOF Bridge is developed and implemented on the basis of the model interface. It extends the validation framework to support OCL constraints defined on MOF compliant metamodels. Chapter 8 refines the MOF Bridge to support the varying versions of the UML metamodel and UML Profiles. Chapter 9 concludes with a summary and gives a short outlook to OCL 2.0 [ocl04].

1.3 Typeface Conventions

This thesis uses the following typeface conventions:

- `production rules, terminal and nonterminal symbols`
- `Java code`
- OCL constraints and expressions

Chapter 2

Assumptions about the OCL

A number of inaccuracies and inconsistencies within the OCL standard impact on the development of an OCL validator. This chapter introduces and lays out the assumptions made in design and implementation of the interpreter.

2.1 The Grammar

A grammar is defined as a 4-tuple (Σ, N , S , P), where Σ is an alphabet, $N \subseteq \Sigma$ is a set of non-terminal symbols, $S \in N$ is a start-symbol and P is a set of productions of the grammar. The OCL grammar is not explicitly defined; only the sets of productions and non-terminals are specified [OCL03, cha 3.9]. We assume:

1. The start-symbol of the grammar is the non-terminal `oclFile`.
2. Σ corresponds to UNICODE [Con03].

2.1.1 Package Construct

[OCL03, cha. 3.5] states that: "[. . .] constraints can be enclosed between 'package' and 'endpackage' statements". Regarding the derivation of the nonterminal `oclFile` with

```
oclFile := ( "package" [...]  "endpackage" )+
```

we conclude that the package-endpackage statement is mandatory and not optional as stipulated by the statement above.

2.1.2 String Literals

The predefined type String is specified as the representation of a string consisting of ASCII or multi-byte characters [OCL03, cha. 8.1.9]. The corresponding production of the string literal is defined as:

```
string := "'"
          (( ~ [ "'" , "\\", "\n", "\r"] )
            | [...]  )*
          "'"
```

The OCL standard states that the Java-CC [JCC03] notation for the symbol '$\sim$' is used. Java-CC defines: "If the character list is prefixed by the '$\sim$' symbol, the set of characters it represents is any UNICODE character not in the specified set." [JCC03, URI: `/doc/javaccgrm.html`].We adopt the definition of the grammar and extend the OCL Type String to UNICODE. This extension can be used to support international applications [ISS03].

2.1.3 Boolean Literals

In [OCL03, cha. 4] the values true and false are introduced as type Boolean. The literal false is then used as an example in [OCL03, cha. 4.4] in the OCL expression: 23 * false. As a subexpression the literal false (and supposedly true) is regarded as an OCL expression itself. We therefore extend the production `literal` through the choice and production `boolean`:

```
literal := string | number | boolean | enumLiteral
boolean := "true" | "false"
```

2.2 Isolation of the Model

We regard every model in the context of OCL as closed. Thereby each property defined in the model can only be parameterized by and result in types contained in the model itself.

2.3 Denotational Character of Qualifiers

A consequence of the closeness of the model is that navigation cannot be qualified via predefined types. The following example of OCL would therefore not be defined in its context [OCL03, cha. 5.7]:

```
self.customer[8764423]
```

In the context of OCL the qualifier of this navigation is a number and thus of the basic type $\mathrm{Integer}$. This argument is not part of the model and the expression would therefore be undefined.
In order to support these types of qualifications we assume that qualifiers are used as denotations. Thus qualifier expressions are type-checked and evaluated; the result is a denotation that must be interpreted by the model.

2.4 AllInstances

According to the type definition of $\mathrm{OclType}$ the feature $\mathrm{allInstances}$ is an operation. In [OCL03, cha. 5.11] this property is used as a structural feature. We assume that the type definitions are generally more precise than the examples and therefore interpret $\mathrm{allInstances}$ as operation.

2.5 Ambiguous Navigation Caused by Shorthand for Collect

As described in [OCL03, cha. 5.7], an association with multiplicity one or zero can be navigated to another single model element. This single model element can also be used as a one-membered collection. A collection's property can be accessed via the shorthand for $\mathrm{collect}$ [OCL03, cha. 6.2.1]. Suppose a property is addressed as the result of such a navigation, it could represent either the property on the single model element or the shorthand notation for $\mathrm{collect}$ on the collection. Though the result seems similar, the type is ambiguous; it may be the result type of the property itself, or a Set of the result type of the property. Oddly an example can be found in the OCL standard itself (refer [OCL03, p. 6-8]):

```
context Person inv:
    let income : Integer = self.job.salary -> sum()
    ...
```

In this example a person is navigated to its job. The result of the navigation is a job acting as a $\mathrm{Set}(\mathrm{Job})$. The example suggests that the property salary should then be collected. The result type would be a $\mathrm{Set}(\mathrm{Integer})$ which defines the collection property $\mathrm{sum}()$. The fact that the model element always defines the feature itself is not considered. Its result type would be of type $\mathrm{Integer}$; the property $\mathrm{sum}()$ is undefined.

In order to avoid this ambiguity result types of property calls after navigation over an association with multiplicity one or zero are considered as a model element acting as a collection including itself. In the example above this convention causes that the attribute salary is handled as an association. The multiplicity is one or zero and the result type is thus defined in terms of OCL.

2.6 Scopes

Some scopes of variables are not defined within the standard, it is assumed that:

- The scope of iterator-variables is the property call, the variables are declared in [OCL03, cha. 6.7].
- The scope of invariant or context names is the whole file.
- The scope of result and parameters used in an operational context is the context itself.

2.7 Exceptions from Undefined Values

Generally, when a subexpression in OCL results in an undefined value the expression itself results in an undefined value. The binary operation or constitutes an exception. If one of its subexpressions results in true, the whole expression will result in true. In order to represent the logical equivalence of $\mathrm{a\ implies\ b}$ to $\mathrm{not\ a\ or\ b}$, where a, $\mathrm{b} \in \mathrm{Boolean}$, we have to extend the originally defined list of exceptions. Therefore we define that the operation $\mathrm{implies}$ always results in true, if a is false or b is true. This will especially apply if the other subexpression is undefined.

With these clarifications we have a sound basis upon which we build our OCL interpreter. We will have to add a minor issue regarding recursion in definition constraints detailed in chapter 5, which we will postpone here for the sake of clarity.

Chapter 3

Accessing the OCL Context

> "Each OCL expression is written in the context of a UML model, a number of classifiers (types/classes, ...), their features and associations, and their generalizations. All classifiers from the UML model are types [...]" [OCL03, p. 8]

OCL is a part of the UML specification. As such, OCL is described in the context of UML models. The semantics of its concepts and constructs is founded on the UML metamodel and its elements. To extend this definition and validate OCL constraints on arbitrary models, this chapter presents a minimal interface querying the important OCL-related information. We therefore dwell on the subset of UML metamodel elements used in OCL. An implementation of the interface specifies the similarities of a model and the UML metamodel. This minimal relation of UML and OCL is throughout used. It suffices for the interpretation of model semantics in the context of OCL.

Requirements relate to the design of either types or instances and are grouped accordingly in the interface description. The chapter concludes with a discussion about design decisions.

3.1 Importing a Model into the OCL Context

The basic idea of extending OCL constraints for a non-UML model is based on the introduction of a subset of UML concepts in OCL. This subset basically consists of classifiers, features and instances that are directly related to the corresponding OCL model types, properties and instances. If we relate the elements of this subset to the elements of a specified model, we can transitively relate the concepts of OCL to those of that model. The resulting mappings define a conceptual binding between the model and OCL as depicted in figure 3.1.

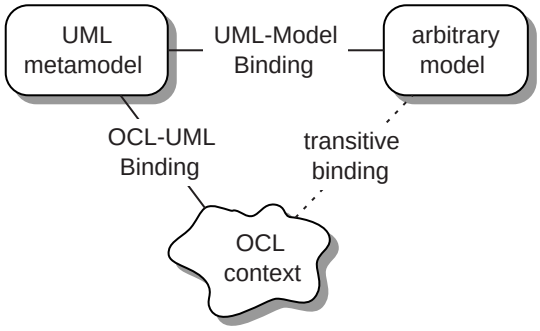


Figure 3.1: The Conceptual Model-OCL Binding represents the mapping between concepts of a model and OCL

Overall the conceptual binding can be used to interpret and validate OCL constraints on the model. We regard the model as being imported into the OCL context.

3.2 Interface Specification Conventions

We describe our Java interface specifying the requirements defined in [API]. In the succeeding sections, we concentrate on the description of the included interfaces and methods. If not otherwise specified, the following will hold in general:

State Information and Transition The state of the model (interface) does not change, it is instantaneous.

Security Constraints There are no security constraints.

Range of Valid and Null Argument Values Except the value `null`, all parameter and return values are valid.

OS/Hardware Dependencies and Serialized Form Section 3.5.2 discusses details of operation system and hardware dependencies and serialization.

Allowed Implementation Variances Allowed implementation variances are addressed in section 3.5.1 .

3.3 Type Level Requirements

OCL constraints are built around objects and object properties. Since OCL is a typed language, each of these objects has a type. Types are grouped in predefined types and user-defined model types [WA99].

3.3.1 Classifier

The predefined OCL types, especially the type OclAny, represent the foundation of model types:

> "Within the OCL context, the type OclAny is the supertype of all types in the model and the basic predefined OCL type." [OCL03, p. 30]

We use this generalization to integrate a model classifier into OCL analogously to a UML Classifier (s. Listing 3.1, line 2). The predefined behavior of the model's basic types, enumerations and other types can then be identified and distinguished by their OCL supertype. Due to type conformance an assignment of a model's Boolean type B to its predefined type describes for instance, that if-clauses can also be parameterized by B in an OCL expression. Transitivity of the type conformance hierarchy inside the model is established through the declaration of each classifier's supertypes (s. Listing 3.1, line 3).

State and Behavior Traditionally an object is an entity that is defined by state and behavior [RM-95]. In OCL states are handled by UML state-machines that are attached to classifiers. The set of states defined on a classifier is represented through the Operation `Classifier.getStates()` (s. Listing 3.1, line 5). The states of an instance can in return be queried in an OCL context with the operation $\mathrm{oclInState(OclState)}$ [OCL03, cha. 5.10].

The behavior of OCL objects is specified by properties or in terms of the UML by features. Each classifier must declare the set of accessible properties it defines (s. Listing 3.1, line 4). For further information regarding properties see the following section which describes them in more detail.

```
1 public interface Classifier {
2     String getPredefinedSupertype();
3     Classifier[] getSupertypes();
4     Feature[] getFeatures();
5     String[][] getStates();
6     String getName();
7     boolean equals(Object type);
8 }
```

Listing 3.1: Java interface describing a model classifier

Listing 3.1 shows the Java interface describing a classifier for OCL. In addition to the type hierarchy required to determine type conformance (line 2 and 3), the behavior and eligible states of each of its instances (line 4 and 5), identity (line 7) and naming context (line 6) are declared.

Enumeration

As a special classifier an enumeration can be directly addressed in an OCL context [OCL03, cha. 4.2]. An enumeration type must therefore be further refined through the specification of its labels (s. Listing 3.2). A label can then be used to resolve a specific enumeration from its enumeration type. The label is used in both the type checking and the evaluation process. The type checker must assert that each

enumeration and enumeration type is defined. The evaluator uses the label to build a specific instance. For further information see chapter 5 and 6.

```
1 public interface Enumeration extends Classifier {
2     String[] getLabels();
3 }
```

Listing 3.2: Java interface describing an enumeration type

3.3.2 Properties

There are three categories of properties in OCL: attributes, associations and operations. Each of these properties is specified and therefore identified by its name (s. Listing 3.3, line 3). In general the name must be unique within the property's namespace (e.g. its classifier). In case of operations the signature must additionally be considered.
The specified name can then be used to resolve a property on a classifier. As we observe in chapter 5, the OCL model provides more than one level, on which types can be instantiated. Features can therefore be described on the classifier or instance level. To distinguish between class-scoped properties and properties that are used on instances of classes, a property must declare its scope (s. Listing 3.3, line 4). It is either specified by the constant `"instance-level"` or `"classifier-level"`. The scope can then be used to check if the property is accessible by specific instances or types.

Attribute

To describe a simple attribute the type of the property must be specified (s. Listing 3.3, line 2). The type information is necessary because every OCL expression evaluates to a specific object of a specific type. Since the model is closed this type has to be located in the model and cannot be predefined (cmp. chapter 2.2).

```
1 public interface Feature {
2     Classifier getType();
3     String getName();
4     String getScope();
5 }
```

Listing 3.3: Java interface describing a property

Listing 3.3 shows an interface which abstracts a property. The property is specified by its name (line 3), has a type (line 2) and is defined either on the instance or classifier level (line 4).

Association

Associations are special properties in OCL. We treat associations rather as their association ends than as the associations themselves. Instead of defining associations in its set of properties, a classifier specifies association ends one may navigate to in an OCL constraint. Thus a navigable association end is identified by its role name.

Navigation of such association ends can result in a single instance being treated as a collection [WA99, cha. 3.6.1]. This occurs when associations are navigated via association ends with multiplicity one or zero. Otherwise navigation results directly in a collection. In order to make the result type distinguishable, a model must provide the declared upper multiplicity of an association end (s. Listing 3.4, line 2). The collection obtained, can however be either a bag, a set or a sequence. A sequence will only result, if the association is ordered (s. Listing 3.4, line 3).

```
1 public interface AssociationEnd extends Feature {
2     int getUpperMultiplicity();
3     boolean isOrdered();
4 }
```

Listing 3.4: Java interface describing an association end

Listing 3.4 shows the abstraction of an association end. The association end is treated as a property of a classifier. It defines its upper multiplicity (line 2) and whether it is

ordered or not (line 3). This information is used to distinguish between collections, sequences and instances acting as one-membered collections.

Operation

Operations as OCL properties are side-effect free query operations. This implies that the set of operations of a classifier only contains queries. As part of the UML an OCL operation of this set is specified through the well-formedness rules of the type Operation [UML03, p. 2-54]. These rules distinguish operations by name and signature. This means that the number, kind and order of the operation's input parameters must be additionally specified (s. Listing 3.5, line 2). The return types of operations are not considered.

```
1 public interface Operation extends Feature {
2     Classifier[] getParameters();
3 }
```

Listing 3.5: Java interface describing an operation

Listing 3.5 specifies a query operation of a classifier. Besides the features of an abstract property the number, kind and order of the operation's parameters are declared (line 2).

3.3.3 Accessing Types

OCL contexts are, as defined in chapter 2.1.1, braced in package-endpackage statements. These specify which package a classifier or a constraint belongs to. The existence of the package in the model's package hierarchy can be checked via the TypeFacade (s. Listing 3.6).

```
1 public interface TypeFacade {
2     Package[] getAllPackages();
3 }
```

Listing 3.6: Java interface describing an access point to the type level

A package can then be resolved through its qualified name (s. Listing 3.7 line 3). In combination with the `Package` interface the `TypeFacade` provides an access point to the model's type system from the perspective of OCL (s. Listing 3.7, line 2).

```
1 public interface Package {
2     Classifier[] getContents();
3     String[] getQualifiedName();
4 }
```

Listing 3.7: Java interface describing a package

3.4 Instance Level Requirements

OCL constraints are contextually defined for types. We assume that a constraint defined on a type will be satisfied if the constraint is satisfied for every instance of that type. To validate constraints an evaluator therefore has to access the instance level of a type.

3.4.1 Access to the Instance Level

In OCL the extent of instances of a type is defined as "[...] the Set of all instances of the type in existence at the specific time when the expression is evaluated" [OCL03, p. 19].

A model can express its instantaneous state and own extent for example to a particular package through the method `ModelFacade.getAllInstances(Classifier)` (s. Listing 3.8). Overall a `ModelFacade` is a description of a model complying to the requirements needed to import a model into the OCL context.

```
1 public interface ModelFacade extends TypeFacade {
2     public Instance[] getAllInstances(Classifier type);
3 }
```

Listing 3.8: Java interface describing a model facade

3.4.2 Instance

An instance of a model type inherits the behavior of its predefined parent in order to fulfill the required type conformance rules [OCL03, cha. 6.4.4]. These encompass the operations defined by the type OclAny. Although a more general static type of the instance is given, neither its specific runtime type nor the nature of its predefined parent is known.

Runtime Types We take a look at the following more general declaration of an OCL constraint defined on the UML metamodel:

```
context ModelElement inv:
    ...
```

The specified constraint is defined for all elements in the UML metamodel. The static model type ModelElement of corresponding model instances is specified. But to evaluate the constraint in the correct context, for example in the context of a UML Classifier or a UML Feature, the runtime type of these objects must be known.

We therefore discuss the operations of OclAny in detail:

oclIsTypeOf(OclType) This operation and the operation $\mathrm{oclIsKindOf(OclType)}$ cannot be implemented using the static type. We have to assert that instances provide access to their actual runtime type (s. Listing 3.9, line 2).

oclAsType(OclType) The argument of the operation must be a supertype of the static type of the instance. The type can be resolved; its property definitions directly accessed. These must then comply to the rules of this operation, i.e. dynamic binding may not be used.

oclInState(OclState) If a state-machine is supplied for an instance, an OCL tool will implement the operation $\mathrm{oclInState(OclState)}$ as defined. The state must then be accessible (s. Listing 3.9, line 3).

Equality Operation ('=') The '$=$' operation and therefore its counterpart '$<>$' are directly mapped to the the Java primitive method `equals(Object)` [j2s03] (s. Listing 3.9, line 4).

Other Predefined Parents In case an instance is not directly based on OclAny the nature of its predefined parent is unknown and cannot be determined via the information the type level provides. An example can be found in figure 3.2. It depicts

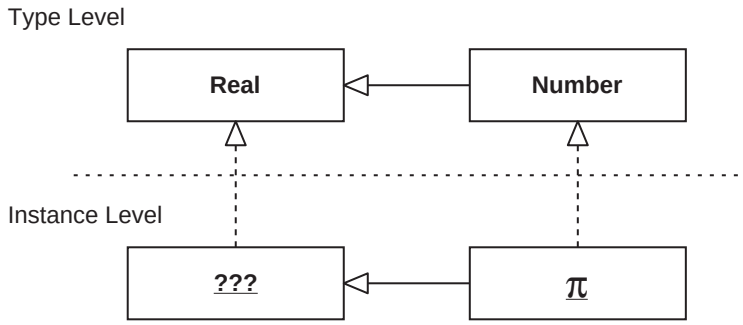


Figure 3.2: An example of an undetermined parent

a number π that is related via its type and the conceptional model binding to an instance of the predefined OCL type Real. To inherit the predefined behavior of its predefined type, the nature of this instance must be known. But neither the model instance's name nor the type level reveal it.
Therefore every model instance has to specify its predefined parent (s. Listing 3.9, line 5). The representation of the parent is further discussed in section 3.5.2.

Values of Properties To access the values of properties we add the specification of properties to the instance itself. This allows the separation of type checking and evaluation. Attributes, associations and operations can therefore be accessed directly through the instance (s. Listing 3.9 line 7 - 10). If the properties are not defined on the corresponding type or their execution causes any form of `Exception`, an `UndefinedFeatureException` and `FeatureProcessingException` will be thrown respectively. These exceptions are left out in Listing 3.9 for the sake of conciseness.

Names of Instances The name of an instance is a `String` representation that can be used by OCL tools for reports etc. (s. Listing 3.9 line 6). In case of an enumeration value it is equivalent to its label. Otherwise an enumeration cannot be

resolved from its enumeration type when directly specified within an OCL constraint.

```
1  public interface Instance {
2      Classifier getRuntimeType()
3      boolean isInState(String[] state);
4      boolean equals(Object instance);
5      Object getPredefinedParent();
6      String getName();
7      Instance getValue(Feature attribute)
8      Instance invoke(Operation operation, Instance[] args)
9      Instance[] navigate(
10         AssociationEnd role, String[] qualifiers)
11 }
```

Listing 3.9: Java interface describing a model instance

Listing 3.9 shows a model instance from the viewpoint of OCL. It describes its predefined parent by specifying its nature (line 5), its state (line 3) and type (line 2). Combined with its identity (line 4) the predefined parent, in particular the OCL type OclAny, can be implemented in an OCL tool. This predefined parent is then extended through the specification of the instance's model features (line 7 - 10). Eventually the name of the instance is specified to allow reporting and the construction of enumerations (line 6).

3.5 Design Decisions

During the development of this interface several design decisions are made. We discuss the qualities and the implementation of these choices.

3.5.1 Rejection of Redundancies

The interface does not contain any superfluous repetition. This affects the performance of an implementation. Therefore this chapter should not be understood as

an approach to describe an interface for practical usage. The described interface is rather a minimal set of requirements a model must comply to in order to make use of OCL in its full extent.

3.5.2 Implementation Independence

One of our objectives is to describe the interface in an implementation independent way. We accomplish this by using primitive types like `String`, `Boolean` or `Integer` and their compositions in our description. This also relates to the representation of predefined types and instances.

Specification of OCL Supertypes Each classifier specifies its predefined supertype (s. Listing 3.1, line 2). The import of model classifiers in the context of OCL is therefore restricted to single inheritance. This restriction for primitive types of the model is based on the presumption that models do not define mixed OCL type equivalents. This allows an OCL tool implementer to resort to an arbitrary OO language environment.

The Method `Classifier.getPredefinedType()` describing the predefined parent of the model type is of type `String` (s. Listing 3.1). The result is defined by the supertype's unique OCL name (s. [OCL03, cha. 8]).

Specification of OCL Parents A model instance must specify the nature of a predefined parent. Types of instances map to predefined parents as defined in table 3.1. The representation can then be used to fully implement the predefined part inherited by a model instance. Overall the described interfaces are structured primitive types. The interface can therefore be considered serializable, platform and implementation independent.

Instance of type	Native representation
Boolean	Boolean
String	String
Real	Double
Integer	Integer
OclExpression	String
OclState	String[]
Enumeration	String[]

Table 3.1: Mapping of instances from OCL to Java

3.5.3 Evaluation Independence

Overall this interface definition is based on the semantics of the model and is independent of the evaluation of OCL expressions. This implies that implementations of the model interface conform to OCL. In comparison to the model type implementation in [Fin00] this is an improvement in correctness and implementation cost of a model in an adaptable OCL evaluator.

3.5.4 Separation of Type and Instance Level

The interface strictly separates the type from the instance level. This allows an OCL tool to use a stand-alone context or type checker in case the evaluation of constraints is omitted.

3.6 Correctness of the Interface

In order to prove the interface conceptually the implemented OCL interpreter makes use of a similar interface. This interface uses controlled redundance to enhance performance and reduce implementation cost. However, both interfaces use the same amount of information. As we observe in chapter 5 and 6, the information is sufficient to implement an OCL validator. We therefore conclude that the interface is minimal

and meets its requirements.

Summary

In this chapter we have successively build a model interface from its type to its instance level. We have shown that the interface provided is minimal and complete regarding requirements of OCL to access concepts of a model. We will prove the interface to be sufficient for the context checking and evaluation process by the OCL validator reference implementation developed in this thesis.

Chapter 4

Parser

This chapter discusses choices of parser generator technologies to be used in the implementation of this framework. An initial examination of the grammar provided in the OCL specification reveals that it belongs to a class of grammars unsuitable for most parser technologies. We then choose a compiler generator balancing the need for changes to the grammar with re-use qualities of the final product.

4.1 Parser Families

In the following we assume that the reader is familiar with grammar basics. That is: regular expressions, context-free grammars, Extended Backus-Naur Form (EBNF). Quoting [ASU99] we shortly introduce the two main categories of parsing algorithms, Bottom-Up and Top-Down Parsing:

> The basic idea of **Bottom-Up** parsing is to begin with the concrete data provided by the input string – that is, the words we have to parse – and try to build bigger and bigger pieces of structure by reducing corresponding right hand sides of production rules to its left hand sides. The building process of an abstract syntax tree (AST) begins at its leaves and is constructed bottom-up to its root, hopefully the start symbol of the

grammar.

Inversely, a **Top-Down** Parser begins its parse process at the start symbol as the root and tries to build the abstract syntax tree by unfolding its production rules until the produced word matches the input string.

In both cases the basic problem is choosing which production rule or production rule alternative to use at any stage during a derivation. To make implementation easier, we introduce **lookahead** as a means of attempting to analyse the possible production rules that can be applied, in order to pick the one most likely to derive the current symbol(s) on the input.

This quality criterion sorts parsers into algorithm families based on their parsing method: Top-Down parsers belong mainly to the LL(k) family and Bottom-Up parsers to the LR(k) family.

Parsers of the LL(k) family scan input left to right and derive the parse tree by its leftmost nonterminal with k tokens of lookahead. LR(k) parser scan input from left to right but conversely derive its tree by its rightmost nonterminal symbol with k tokens of lookahead.

4.2 Characteristics of the OCL Grammar

To benefit from a wide range of parser generators, a language or rather its grammar should be at least LR-parseable. Unfortunately this property does not apply to the OCL grammar.

4.2.1 Path Name Ambiguity

In the OCL grammar an ambiguity between path names is caused by the production rules:

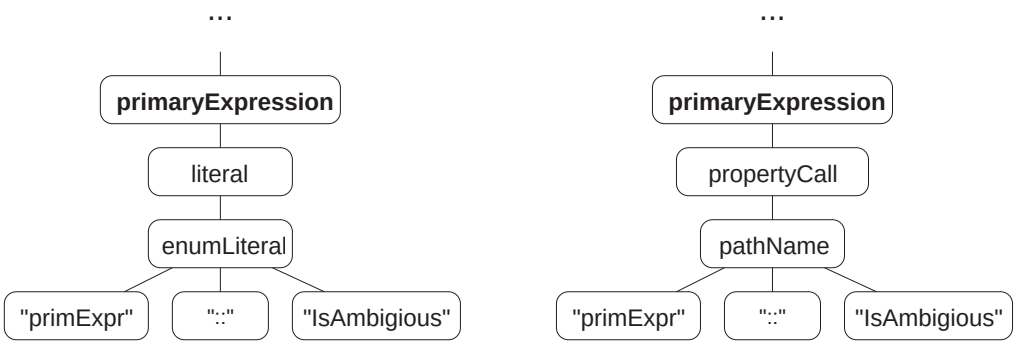
```
pathName := name ("::" name)*
enumLiteral := name "::" name ("::" name)*
```

For this reason the grammar in its original form is inappropriate for most parsing technologies.

Proof by Example. To show that the OCL grammar is not LR-parseable we show that the grammar is ambiguous [ASU99]. An ambiguous grammar, related to a parsing algorithm, is defined as a grammar that allows derivation of more than one parse tree for a specific input. We study the input:

```
package proof
    context Example inv:
        primExpr::IsAmbigious
endpackage
```

This is clearly legal OCL and fits into the following two outlined parse trees:



Thus the grammar of OCL is, regardless of the order of the lookahead, unparseable by LR-algorithms. □

4.2.2 Name-Expression Collision

The next problem is based on the fact that the nonterminal `expression` can be derived to the nonterminal `name` as depicted in figure 4.1.

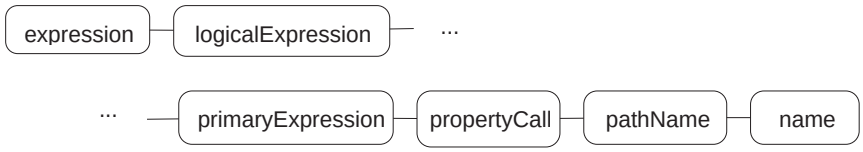


Figure 4.1: The derivation of an `expression` to a `name`.

Consider the OCL production rules:

```
propertyCallParameters :=
    "(" declarator?  actualParameterList?  ")"

declarator :=
    name ( "," name)*
    (":" simpleTypeSpecifier)?
    (";" name ":" typeSpecifier "=" expression)?
    "|"

actualParameterList := expression ( "," expression)*
```

In this set of production rules the production `propertyCallParameters` is ambiguous for a parsing algorithm part of the LR(k) family with a fixed lookahead of k.

Proof by Contradiction. Take $k \in \mathbb{N}$ to be a fixed number, n_i to be derivable by the production rule `name` for all $0 \leq i \leq k$ and $w \in$ OCL to be an input of the form:

```
package proof
    context Contradiction inv:
        ModelElement::propertyCall(n0, n1, ..., nk)
endpackage
```

Suppose $lr \in \text{LR}(k)$ to be an algorithm unambiguously parsing the OCL grammar. To simplify this, let us assume that lr is a Bottom-Up parser, without loss of generality.

We use induction on k to show the contradiction for every $k \in \mathbb{N}$:

Consider that $lr \in \text{LR}(0)$ tries to parse the word w. Scanning through the input with $k = 0$, lr recognizes after iterative derivations n_0 which corresponds to the right hand side of the production rules `declarator` and `actualParameterList`. With no lookahead available lr clearly cannot decide which production's right hand side should be reduced.

We assume that a LR(k)-algorithm cannot unambiguously derive the input n_0, n_1, ... , n_k from the production rules `declarator` and `actualParameterList`. By induction we conclude that consequently any $lr \in \text{LR}(k+1)$ cannot unambiguously parse n_0, n_1, ... , n_{k+1} with $k+1$ tokens of lookahead. Since the next token `","` or n_i of the input n_0, n_1, ... , n_{k+1} can be derived from both, `declarator` and `actualParameterList`, the additional token of lookahead does not bring a decision.

The result is that for every fixed $k \in \mathbb{N}$, $lr \in \text{LR}(k)$ cannot parse the changed OCL grammar unambiguously. □

For the sake of completeness we note that consequently the changed OCL grammar is thus also not LL(k), in particular not LL(1)-parseable, since LL-languages are a subset of LR-languages [Pep97].

4.3 Improvements towards an LALR(1)-parseable Grammar

In order to adapt the characteristics of the OCL grammar to our needs, we change the grammar without changing the language itself.

By removing solely the production rule `enumLiteral` the ambiguity of path names described in section 4.2.1 is solved. Also, originated from the production rule literal, an `enumLiteral` is then recognized by a parser through the nonterminal `pathName` of a `propertyCall`.

Further changes to the grammar at this point relate to the core `expression` production rule of the grammar (s. section 4.2.2).

Existing OCL parsers for OCL 1.3 – for example the IBM parser reference imple-

mentation of the OCL specification [IBM03] or a parser framework build at the TU Dresden [Fin00] – circumvent this problem in different ways:

The IBM parser was generated using the JavaCC compiler generator. Its grammar file shows that the production rule `featureCallParameter`, later renamed `propertyCallParameter` in OCL 1.4, is defined using the JavaCC feature 'LOOKAHEAD'. This feature establishes a default lookahead of 2147483647 tokens. One could argue that practically there won't be cases where this number could be exceeded, but theoretically this solution is not only incorrect but also not LL(1)-parseable, as originally stated in the OCL specification.

The TU Dresden parser, employing on the fact that names are not structurally distinguished from expressions, uses expressions for names. The production rule changes to:

```
declarator :=
    actualParameterList (":" simpleTypeSpecifier)?
    (";" name ":" typeSpecifier "=" expression)?  "|"
```

This grammar still recognizes all OCL language constructs.

A parser based on this grammar accepts a larger language than the original grammar allowed. The TU Dresden Parser makes use of the SableCC parser generator framework [Gag98]. This framework is able to accommodate the workaround introduced above. Using its *ignored alternatives* [Gag98, cha. 5.3], we can instruct it to skip the original problematic production rule alternative in the grammar, but to establish its node usable for the parsing tree. The initial concrete syntax is built into a parsing tree. This concrete syntax can still express a larger language than OCL. In a second phase, the parsing tree is transformed into an abstract syntax tree. During this phase the offending constructs are identified and rejected.

SableCC produces parsing trees that are based on the Java type system. It additionally provides a representation of the *visitor* in combination with the *adapter* design pattern [GHJV95]. This allows depth first traversal of the tree [Gag98, cha. 6.4]. We can easily extend the original parser to transform the concrete syntax tree into the correct abstract syntax tree. This is achieved by overriding the *before methods* of the

adapter (for more details s. cha. 5.2.2). During the parsing process these are called by the framework each time before a node of the tree is visited, to check colliding `expressions` in order to replace them by the expected `name`'s node. The result is an AST compliant with the original grammar of OCL.

Comparing the solutions of the IBM parser reference implementation and the TU Dresden parser, we can conclude that the TU Dresden approach is more efficient, more stable and so overall more feasible for our problem.

A SableCC version of our grammar can be found in Appendix A. It has been translated using SableCC and thus is LALR(1) and thereby, since languages based on LALR(1)-parseable grammars are a direct subset of these derived by LR(1)-grammars, LR(1)-parseable [ASU99].

Summary

In this chapter, we have discussed an analysis of the OCL grammar for aspects of usability with existing parsing technologies. We have determined that the grammar is not LR-parseable regardless of the number of lookaheads. Removal of the ambiguous production rule `enumLiteral` has eliminated the first cause of the grammar's incompatibility to LR-parsers. Comparison of existing OCL parsers has led us to work around the second cause using the SableCC framework. The production rule `declarator` has been changed to encapsulate a name into an expression. The resulting concrete syntax tree can be converted to an abstract syntax tree by using the visitor pattern provided by our chosen parsing framework. Thus the original OCL grammar that is not even LR-parseable has turned into an LALR(1)-grammar.

Chapter 5

Context Checker

The next component in the pipe and filter pipeline introduced in chapter 1 is the context checker. The context checker controls the use of OCL constructs within the abstract syntax tree produced by the parser. In contrary to the parser, this examination is not solely based on the syntax, but also on semantics of OCL.

We start by identifying context dependent language features. They correspond to contextual tests which the context checker implementation must contain.
We further proceed to the primary function of the context checker – the type check. Before we start to explain the type checking process itself, we analyze the type system of OCL focusing on its representation in the Java type system [GJS96]. We use the Java Reflection API to obtain information about predefined types represented by Java interfaces. Changes to these interfaces allow us to adapt and reflect this part of the type system within our context checker. Combined with model types described in a `TypeFacade` (refer chapter 3.3.3) the implementation of the OCL type system is completed. The implementation is used to present a type checking algorithm that is based on the abilities of the chosen parser framework.

5.1 Context-dependent Constructs

Constraints included in this list are more or less directly addressed in [OCL03]. Facing the inaccuracies of the OCL standard found in the previous chapter 2 and 4 it is conceivable that other exceptions can occur. The list therefore does not claim to be complete.

5.1.1 Reserved Keywords

Reserved keywords (e.g. if or $\mathrm{context}$) cannot be used as names of packages, types or properties [OCL03, cha. 4.8]. Declaring these keywords as tokens in the SableCC grammar enforces this constraint. It is implied that all tokens within the grammar must be handled as reserved. A complete list of tokens, including for instance the added tokens `true` and `false` (cmp. chapter 2.1.3), can be found in the provided SableCC grammar (s. Appendix A).

The grammar does not define the keywords self and result as tokens. In order to provide a similar treatment without changing the grammar, self and result must be controlled to be directly derived from the nonterminal `propertyCall` defined by the production rule `primaryExpression`.

A specified property call must in return be a single name (i.e. plain self or result). This implies that the property call must not be specified by a qualified path name and must not define the time expression $@\mathrm{pre}$, nor qualifiers or parameters. In case of result the primary expression must additionally be included in a postcondition.

5.1.2 Primary Expressions

The nonterminal `propertyCall` in the production rule `primaryExpression` is used to specify many different entities and features of OCL. Besides an operation call or a call on attributes or associations, the production rule `propertyCall` is used to describe variables, model types, states, enumerations and the keywords self and result. Accordingly, this production provides many optional nonterminals that are

contextually limited concerning the compatibility to individual entities. For this reason the context checker has to check that parameters are used for operations and qualifiers for associations only. Property calls may use the time expression $@\mathrm{pre}$. This and the operation $\mathrm{oclIsNew}()$ must then be used exclusively in postconditions. In contrast to model types, states and enumerations, the names of properties and variables may not be qualified additionally.

5.1.3 Arrow and Dot Operators

The value of a property of an object is specified by a dot followed by the name of the property. A property of collections is accessed by using an arrow followed by the name of the property. To satisfy that both operators are used appropriately, the type of an object has to be determined by the type checker first. Then the use of the operator can be checked.

5.1.4 Declarators

"All collection operations with an OclExpression as parameter can have an iterator declarator." [OCL03, p. 6-36]. The iterator declarator is undefined for other types and operations. This must be asserted by the context checker.

5.1.5 Stereotypes

The stereotype of a constraint must be inv in a classifier context and in operational contexts pre or post.

5.2 Type Checking

> "All types defined in a UML model, or predefined within OCL, have a type. This type is an instance of the OCL type called OclType." [OCL03, p. 29]

This statement describes the overall structure of the OCL type system. It defines a meta, a classifier and an instance-level. The meta level in OCL is represented by the type $\mathrm{OclType}$. Its instances form the classifier level; these are the predefined and model types, whose instances in turn define the instance level.

5.2.1 Implementation of the Type System

In order to let our implementation meet the requirements on this three-tier type system we have to distinguish between the type $\mathrm{OclType}$ and predefined and model types on the classifier and on the instance level respectively.

Predefined Types

To represent the predefined types in our environment we make use of the Java type system. The signature of the predefined types is described in form of Java Interfaces. Our type checker then queries this metadata via the Java Reflection API.

Besides the type system, the advantage of using interfaces instead of plain metadata (e.g. a property file) to describe types lies in the fact that they can be reused in the implementation of instances which we will introduce in chapter 6. By expressing the changes in corresponding interfaces and classes, the validator, in particular its evaluator, after recompilation reflects the modified metadata to its validation process. This also implies that the adaptability of the validator to changes of the type system in later versions of OCL is increased. An important factor of this approach is the power of the Java in comparison to the OCL type system.

Basically we represent an OCL type by a single interface. An interface uses the inheritance mechanism of Java to represent generalization in OCL. The properties of the predefined types in OCL are exclusively operations and can therefore be directly represented by methods, without considering attributes or associations.

Naming Conventions The interfaces are named after corresponding types but prefixed with `Ocl`. This avoids *Name Clashes* with primitive objects from the package

`java.lang`.

The names of operations must be partially changed, due to OCL's use of Java reserved keywords for its operation names. The following mapping is applied for renaming:

OCL name	Java name
$=$	eq
$<>$	neq
$+$	plus
-	minus, negation
$*$	multiplication
$/$	division
$<$	less
$>$	more
$<=$	lessOrEqual
$>=$	moreOrEqual

Table 5.1: OCL operation mapped to Java method names

Redefinitions Another problem of adaptation from the OCL type system to the Java type system is redefinition of inherited operations. In OCL redefinitions are *covariant* [Mey97] as depicted in figure 5.1.

Consider the following example:

```
1 Real r, r2 = ...;
2 Integer i = ...;
3 r = i;
4 r + r2;        // => + : Real x Real -> Real
5 r + i;         // => + : Integer x Integer -> Integer
```

Let the variable `r` have the static type `Real` and the runtime type `Integer` (line 1-3). Although `r`'s static type declares the operation $+ : \mathit{Real} \times \mathit{Real} \rightarrow \mathit{Real}$, it is not defined, because the operation $+ : \mathit{Integer} \times \mathit{Integer} \rightarrow \mathit{Integer}$ is dynamically bound. This is not correct and therefore *covariant* operation redefinitions exclude the

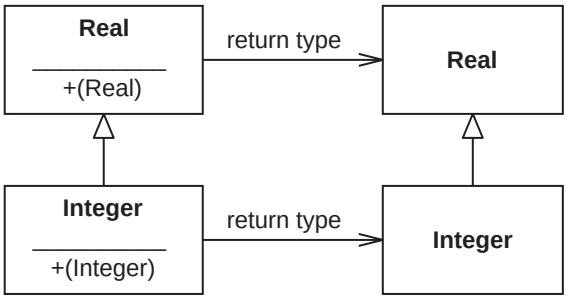


Figure 5.1: Covariant inheritance in OCL

use of dynamic binding as supported in Java.

This implies that we cannot directly use the Reflection API to resolve an operation. Regarding the type hierarchy we determine an operation by an argument's static type. In our example the OCL operation $+$ is resolved according to the types of `r2` and `i` from the interfaces `OclReal` in line 4 and `OclInteger` in line 5.

In addition Java will not allow the specification of covariant return types, if the parameters of an operation are not specialized, as it is the case for the parameterless operation $\mathrm{abs}()$ or '-' from $\mathrm{Integer}$. These return types must be programmatically adjusted. Adaptations to our OCL type system representation are therefore restricted with regard to redefinition: if a return type of an operation is specialized, then at least one parameter type will have to be specialized as well.

Collection Types Collection types in OCL are parameterized types. Since the current Java version 1.4 does not support *generics* in its type system [JSR01], we have to expand the type definition to a class `Type` which at least holds the parameter type of collection types in addition to the type interface. Unfortunately we cannot use this class to represent the type information of the parameter within our meta level interfaces. For instance the declaration of the method

```
Type abs();
```

clearly does not reflect the information whether the return type is actually of type `Integer` or `Real`. Accordingly, signatures of methods, especially their collection parameter and return types, must be programmatically augmented by corresponding

element types. This adds another constraint to our adaptation process which must be addressed in each property resolution. In order to abstract from such augmentations, the type checker internally uses the class `Type` as the uniform foundation of types (s. figure 5.2).

Higher Order Functions Formally Higher Order Functions are defined as functions that are parameterized by functions [Pep99] or in terms of OCL operations that are parameterized by $\mathrm{OclExpressions}$:

> "Each OCL expression itself is an object in the context of OCL. The type of the expression is OclExpression." [OCL03, p. 31]

An example for the use of a Higher Order Function in OCL would be a collection operation call $\mathrm{forAll(\ a,\ b\ :\ Integer\ |\ a <> b\ )}$. The $\mathrm{OclExpression}$ used as parameter in this example includes in addition to the expression $\mathrm{a <> b}$ the optionally declared iterator variables a and b and their assigned type $\mathrm{Integer}$. The evaluation type of the expression is the type of its expression (i.e. $\mathrm{Boolean}$). This implies that each expression in OCL has two types, the expression type and the type it evaluates to. Therefore it is necessary to introduce a new class `ExpressionType` representing the complete $\mathrm{OclExpression}$ as an extension to the class `Type`. This class can then be used by the type checker to store both the expression type and the evaluation type of the expression. To resolve an operation, the type checker then has to query operations twice: the first time as a Higher Order Function (i.e. with the parameter in form of the expression type) and the second time with the evaluated type of the expression parameter. Again, the class `ExpressionType` is an insufficient representation analogous to the representation of collection types. Evaluation types of the parameters of the operations $\mathrm{collect(OclExpression)}$ and $\mathrm{iterate(OclExpression)}$ are used to declare the return type of the respective operation on the meta level. Again, these types must be adjusted programmatically.

In summary we conclude that the Java type system is only partially capable of supporting the predefined types of the OCL type system. The following limitations to the adaptation of our OCL validator must be dealt with:

- OCL names that can cause *Name Clashes* or mangle reserved characters or keywords in Java.
- Redefinitions of the signatures regarding only the return but not the parameter types of an operation.
- Template parameters of collection types in operation signatures.
- Higher Order Functions in form of operations that declare their signature using the evaluation type of the expression parameter.

Existing types and operations are not subject to these limitations. Missing information on the meta level is programmatically adjusted. In the current version of the validator this information is directly encoded in the implementation. Although this is clearly not the best approach, finding a better way is beyond the scope of this thesis. This process is internally hidden through the introduction of the class `Type` and its subtypes.

Model Types

As shown above the class `Type` can be used to hide implementation issues of the type system. We use this to hide the different query techniques used to gather the information of both the model and the predefined types. Therefore we introduce the new subtype `ModelType`. This class adapts a model type to a `Type` using the `Classifier` representation from the model interface (s. chapter 3.3.1).

As the only group of types that defines states and associations, the class `ModelType` must be extended to express the state definitions of the model types and the alternative collection type that will be obtained if an association with multiplicity one or zero is navigated. Additionally the supertype hierarchy of model types must be adapted to the conformance rules of OCL [OCL03, cha. 4.4].

Classifier and Meta Level of OCL

In the OCL context classifiers are objects of type $\mathrm{OclType}$. As such, the properties declared on the type $\mathrm{OclType}$ must be defined for these classifiers. Since these

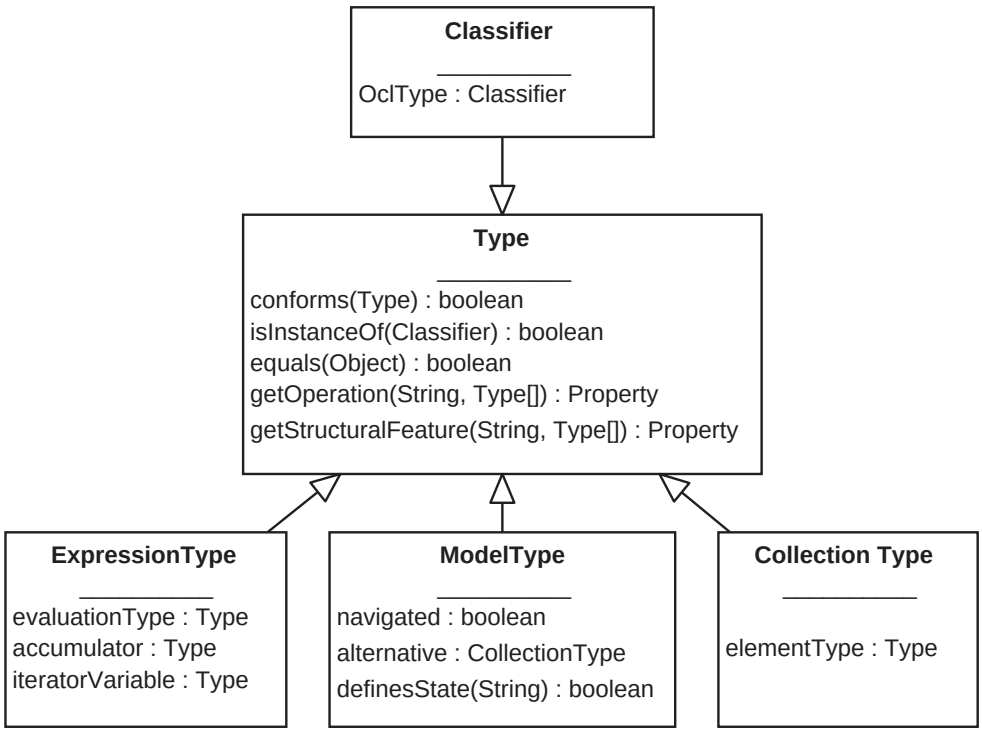


Figure 5.2: The type model diagram depicts the model implementation used in our OCL validator. It includes the classes `Classifier`, `Type`, `ExpressionType`, `ModelType` and `CollectionType`.

properties are not defined for the instances of the classifiers, we have to differentiate between the types themselves and the instances of those types. We therefore introduce a new subclass `Classifier` of the class `Type`. This class represents a type at the classifier level and the class `Type` a type at the instance level. As a special classifier the OCL type $\mathrm{OclType}$ is included in the class definition of a classifier.

5.2.2 Type Checking Algorithm

The type representation introduced above can now be used in our type checking algorithm. The nodes of the AST are labeled with types. If there is more than one type represented by a node, as would be the case in nodes representing operation parameters, types will be labeled in the order of nonterminals in the corresponding production rule. We store evolving labels in our parsing environment so that these are

reusable for the next component in the pipeline of the validator, i.e. the evaluator.

Tree Traversal

The AST is processed with tree-walkers of the SableCC framework. A generated `DepthFirstAdapter` walks the tree in a *depth-first traversal*, calling the respective *before* and *after methods* on entering and leaving nodes [Gag98, cha. 6.4].

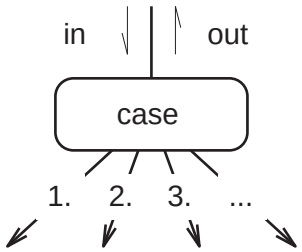


Figure 5.3: The depth-first traversal of a node in the SableCC Framework. The before (in↓), the visit (case) and eventually the after method (out↑) are called for every node from left to right.

These methods cause that the traversal can be partitioned into a descending (in↓) and an ascending phase (out↑). In case this visitation sequence has to be adapted to the requirements of OCL, we override related visit (case) methods and change the order of the tree-walker from left-to-right to an order that fits our needs.

For type checking purposes we mainly visit nodes by their after methods in the ascending phase. Thus the AST is processed bottom-up from left to right.

Cooperation of Visitor Methods

Communication between specific visitor methods – to declare the current type of self for example – is passed along encapsulated instance variables of the visitor. Thereby cooperation between visit, before or after methods is enabled.

Visting a Node

With this skeletal structure of our type checking algorithm we can start at the root of the AST. While descending the nodes, contextual information is gathered to describe the current context of each node. Once the bottom of a branch is reached, it is recursively processed to its root. Each time a node is visited the type label of its children can be accessed from the environment. Combined with the contextual information that is shared with nodes visited before, checks related to the current node can be accomplished. Before ascending to the next node, the current node is in turn labeled with its calculated type.

Application of the Algorithm

In our implementation this is basically the complete algorithm used for type checking purposes. While explaining the individual checks of the nodes is too extensive to be presented here, there are some applications we address in more detail.

Computation of Definition Constraints First Regardless of the specification order in an OCL file, the properties of definition constraints must be declared before evaluating invariant constraints. Therefore the depth-first-order of the type checking algorithm must be adapted. We redefine the visit method of a constraint node to comply with this order.

Primary Expressions Every construct of OCL described by a primary expression must be distinguishable by the type checker. Therefore it must be considered that states, model types and enumerations are all specified by a ::-separated path name list. In an OCL constraint this name list must be unambiguous in each specified classifier and package context of a model. A "good guess" is not supported by our type checker implementation.
In order to make a distinction between a specifier with a list length of one and a feature name, the leading operators . and $\rightarrow$ can be used. Because predefined types do not define structural features, an attribute can then be distinguished from an

unqualified association by querying the model. An operation is especially discerned by the mandatory parentheses specified after the operation's name.

Iterator Variables In contrary to common programming languages, an operation's parameter type is not self-defining in the context of OCL. Each collection operation that is parameterized with an $\mathrm{OclExpression}$ implicitly or explicitly declares an iterator variable representing an element of the collection defining the operation. The parameter type of the collection must be known before the type of this variable and consequently of the $\mathrm{OclExpression}$ can be resolved.
We therefore override the visit methods of `postfixExpressions`. Thus contextual information does not need to be exchanged between affected visitor methods. Each time a property is called, the defining type, the operation name and the parameters are known and can be individually processed by the visit method. This particularly includes declarations of iterator variables. We handle these before further descending into the corresponding $\mathrm{OclExpression}$ parameter node (and thereby losing contextual information).

Recursion in Definition Constraints A let expression allows definition of operations within the OCL context. In a definition constraint let expressions can then be used to extend classifiers by "pseudo-operations". The defined operation is useable in "the same context where any property of the Classifier can be used" [OCL03, p. 8]. This would consequently include the definition constraint itself, but in contrary to recursion in postconditions (s. [OCL03, cha. 5.3]) the use in definition constraints is not explicitly defined. It is unclear if recursion of such operations is intended or the scope of the operation is badly defined. The latter is more likely; otherwise the property could also be used in the same definition constraint before its own declaration.

In keeping with our type checking algorithm, however, we support the recursion of such operations in a limited way: The optional result type of such an operation must be explicitly declared. This does not restrict functionality of the feature, only the means of its declaration. Without this additional constraint the algorithm as described in this chapter would not be sufficient, because the type check heavily relies on mandatory type labels on child nodes. This especially holds for property calls, whose types in their recursive form cannot be resolved until termination. Substantial additional effort in

form of an additional implementation of a backtracking algorithm would be required, if the declaration of an operation's result type was not made mandatory. Therefore we use this restriction without limiting the possible functionality of the recursion.

Summary

In this chapter we have determined the context dependent constructs the context checker must control.

We have developed a type system implementation. Predefined OCL types have been implemented on the basis of Java interfaces and the Java Reflection API. The advantage of this approach is that changes to the predefined types of OCL can be expressed in form of this metadata and in particular in form of its functionality. After a recompilation these can be directly mapped into the type check and the evaluation process of the validator. By comparing the OCL and Java type systems we have concluded that this adaptation process is limited in its expressiveness.

These restrictions have led us to build an abstraction layer in form of the class Type and its subclasses, in order to hide the concrete querying of type definitions from the type checker. The abstraction layer has been in particular useful to integrate our model interface developed in chapter 3. The types can then be represented on the instance, the classifier and the meta level of the OCL type system.

Finally we have used the developed types in a type checking algorithm that is based on type-labeling on respective nodes. The algorithm has been designed employing abilities of the generated parser framework. It is implemented in our context checker, so that labels can be reused by the evaluation component presented in the next chapter.

Chapter 6

Validation of OCL Constraints

OCL constraints are validated against models on the instance level. Specified instances and properties of metamodel types are accessed to evaluate expressed constraints. Evaluation results can be used to validate the correctness of a model. In this chapter an evaluator for OCL constraints is developed. Applicability of compilation and interpretation as alternate evaluation techniques are shortly discussed. The implementation of an interpreter is presented. The model interface is extended to make evaluation results accessible.

6.1 Compiler vs. Interpreter

There are two classes of techniques feasible to evaluate OCL constraints. Compilers compile constraints to code. The code is executable and validates respective constraints against model instances. Interpreters evaluate constraints by processing respective ASTs. For both techniques a finite and immutable description of a model and its instance at a specific moment is required.

Interpreting a constraint is less efficient than the execution of a compiled constraint. But compilation is cost intensive (cmp. [Pep97, cha. 5.1]). Thus interpreters are more suitable for changing or variable constraints. Compilers in contrary are more efficient with respect to fixed constraints.

During the development process of a model, UML Profiles and metamodel constraints are generally static. Thus compilers are actually more efficient in terms of our objectives. But implementation costs involved in such an approach are beyond the limits of this thesis (cmp. [Fin00, cha. 7]). For this reason an interpreter is developed for evaluation purposes.

6.2 Processing the AST

The interpreter processes an AST using the tree-walkers of the SableCC framework (s. chapter 5). The evaluation of invariant, pre and postcondition constraints is discussed. Identified node traversal recursions lead us to an evaluation algorithm similar to the type checking algorithm in chapter 5.2.2. Special issues of the evaluation process are discussed in more detail.

6.2.1 Evaluation of Invariant Constraints

Constraints in OCL are specified as boolean expressions defined in the context of model classifiers. Invariant constraints parameterize these expressions with self (s. listing 6.1). To show the correctness of such a constraint, the expression must be evaluated for every instance of the contextual classifier.

```
context AClassifier inv constraint:
 -- AClassifier.allInstances()->forAll(self | expression)
    expression<AClassifier>
```

Listing 6.1: A parameterized pseudo OCL invariant

On the basis of a finite model it can be assumed that this set of instances is finite. Thus the corresponding set of expressions can be evaluated in finite time.

6.2.2 Evaluation of Pre and Postconditions

A pre or postcondition is additionally specified in the context of an operation. A corresponding expression is parameterized with the contextual classifier and the return or parameter types of this operation respectively (s. listing 6.2).

```
context Classifier :: Operation (P1, P2, ... , Pn): R
 -- Classifier x P1 x P2 x ... x Pn ->
 --  forAll( (self, p1, p2, ... , pn) | expression )
    pre: expression<Classifier, P1, P2, ... , Pn>

 -- Classifier x R x P1 x P2 x ... x Pn ->
 --  forAll( (self, result, p1, p2, ..., pn) | expression)
    post: expression<Classifier, R, P1, P2, ... , Pn>
```

Listing 6.2: Pseudo OCL parameterizing pre and postconditions

The incorrectness of instance models can not be directly inferred from evaluated expression results. A violated pre condition for all possible arguments will not imply the violation within a concrete model instance, if the respective operation is never called with the offending argument. The interpreter evaluates all arguments, though. A model instance will be defined to be correct, if every precondition holds under all possible assignments from well-typed elements of the model. Violations must be individually determined using results with respect to specific arguments.

6.2.3 Interpreter Algorithm

By accessing the instance level via the model interface our instantaneous evaluator obtains the immutable description of model instances. The description is used to interpret constraints with a SableCC visitor. Analogously to the preceding context checker traversal, this visitor generally processes an OCL constraint AST bottom up from left to right (s. chapter 5.2.2). It is recursively executed for all instances of involved parameters as introduced in section 6.2.1 and 6.2.2. The process is supported by the static types and properties determined in the context check.

Analogously to the type system implementation in chapter 5.2.1 we use an abstraction

that hides differences between objects of the model and predefined instances (s. figure 6.1). The resulting interface provides uniform means to access the runtime type, the specific value or among others the properties of an instance (cmp. chapter 3.4.2).

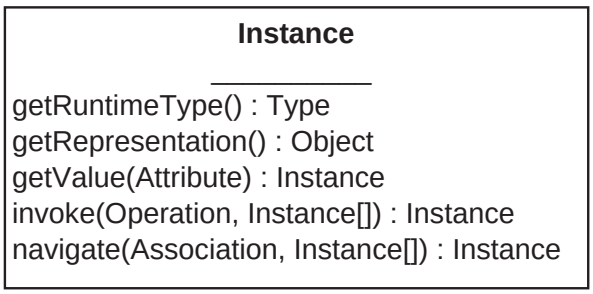


Figure 6.1: The instance abstraction.

During the evaluation process it is recursively used to label instances on respective nodes using our environment. Overall the algorithm is similarly defined to the type checking algorithm in 5.2.2.

6.2.4 Evaluation of OCL Features

Since discussing the evaluation of all OCL features is too extensive, we discuss the evaluation of some more intricate features like Higher Order Functions, undefined values or operation precedence.

Higher Order Functions

As mentioned in chapter 5.2.1 Java does not support Higher Order Functions in its current version 1.4. Thus we have to evaluate these differently from other operations.

In the context of OCL a Higher Order Function is defined on a collection instance parameterized with an $\mathrm{OclExpression}$. The evaluation of the argument expression depends on the context (e.g. the collection's instances, scopes of let expressions etc.). The interpreter stores this contextual information in the visitor's instance variables and

its environment. This information has to be passed to the calling collection instance when Higher Order Functions are invoked (s. figure 6.1). The introduction of the class `ExpressionInstance` circumvents this. An expression instance encapsulates expression evaluation results for individual combinations of iterator variables. These results are calculated in the visitor itself and can be accessed by respective collection instances. A collection instance defines its provided Higher Order Functions with respect to obtained result types.

Example The operation forAll is a Higher Order Function ($*$) in the OCL expression:

$$\underbrace{Set\{1,2,3\}}_{collection} . \underbrace{forAll}_{*}(\underbrace{a,b|a <> b}_{OclExpression})$$

The example's argument expression must be evaluated for all combinations of iterator variables $(\mathrm{a},\ \mathrm{b}) \in \{1,\ 2,\ 3\}$ x $\{1,\ 2,\ 3\}$. The interpreter calculates the expression's results:

a	1	1	1	2	2	2	3	3	3
b	1	2	3	1	2	3	1	2	3
result	false	true	true	true	false	true	true	true	false

The collection instance joins the individual results according to the definition of forAll and returns false. This process is independent of passing contextual information.

Exceptions and Undefined Values

During the evaluation of operations unanticipated exceptions, especially undeclared runtime exceptions, can occur. Respective operation results and expressions are undefined. The interpreter skips the evaluation of the current constraint. Exceptions of undefined expressions for the operations and, or and $\mathrm{implies}$ are reported. Constraints which have been skipped completely are marked as undefined. Because OCL is a side-effect free language, this procedure is feasible.

Enumerations

In OCL enumerations can be specified by name. To resolve enumerations we have to assert that enumeration names are unique within an enumeration type (cmp. chapter 3.3.1). An enumeration is then identified analyzing all instance names of a specified enumeration type.

Precedences

Although operation precedences [OCL03, cha. 6.4.6] do not influence our type check, the evaluation order of combined features is changed. Most precedences are implemented in the grammar [OCL03, cha. 3.9] and thereby are subsequently observed by the evaluation visitor and interpreter. Logical and relational operations pose an exception. We thus extend our grammar with respect to the precedence of the operations $\mathrm{implies}$, $=$ and $<>$. The productions rules `booleanExpression` and `compareableExpression` are added. They are nested within the production rules `logicalExpression` and `relationalExpression` respectively:

```
logicalExpression     := booleanExpression
                         ( impliesOperator
                           booleanExpression
                         )*

booleanExpression     := relationalExpression
                         ( booleanOperator
                           relationalExpression
                         )*

relationalExpression  := compareableExpression
                         ( equationOperator
                           compareableExpression
                         )?
```

```
compareableExpression := additiveExpression
                         ( compareOperator
                           additiveExpression
                         )?
```

This enables the interpreter to still visit the AST bottom-up from left to right without the requirement to sort individual properties.

Time Expression

The time expression $@\mathrm{pre}$ describes values before the execution of an operation. Every operation in the context of OCL is a side-effect free query operation. Thus this feature does not effect the value of an instance and therefore is not considered by the interpreter. The construct is rather understood as a concept for modeling purposes than a feature subject to concrete evaluation.

6.3 Output

Analogously to the abstraction layer of the model, we outline an implementation independent set of information to describe the output of an OCL constraint evaluation.

To access the results of individual constraints, the model interface from chapter 3 is extended. Contexts of invariants can be accessed by model classifiers (s. Listing 6.3, line 2), contexts of pre and postconditions by their operational context (s. Listing 6.3, line 3).

```
1 public interface Report {
2     Context getContext(Classifier classifier);
3     Context getContext(Classifier classifier,
4         Operation operation); }
```

Listing 6.3: A validation report based on the model interface

Individual constraints specified in a context can then be accessed either by its position in the context (s. Listing 6.4, line 2) or if specified by its name (s. Listing 6.4, line 3).

```
1 public interface Context {
2     Constraint[] getConstraints();
3     Constraint getConstraint(String name);
4 }
```

Listing 6.4: A Context interface as a sequence of constraints

The actual results of invariant, pre and postcondition constraints are parameterized (s. 6.2.1 and 6.2.2). The results can thus be queried individually (s. Listing 6.5, line 3) or in general (s. Listing 6.5, line 2). A result is described by one of the three states: `"satisfied"`, `"unsatisfied"` or `"undefined"`.

```
1 public interface Constraint {
2     boolean isSatisified();
3     String[] getResult(Instance[] args);
4 }
```

Listing 6.5: The Constraint interface described as a set of results

In total this extended model interface represents an interface to query OCL validators in an implementation independent way.

Summary

This chapter has developed and implemented an interpreter for OCL constraints. Based on the model interface, the interpreter accesses the instance level of a model to evaluate constraints. Results of interpreted constraints have been represented in a model interface extension. The extended model interface is suitable to validate OCL constraints against models implementation independently.

Chapter 7

The MOF Bridge

Now that we developed an OCL context checker and evaluation component on the basis of an abstract model, we proceed by extending this model to the MOF Model. Therefore we have to align MOF to UML metamodel elements with respect to the OCL concepts. With the resulting conceptual binding we obtain a semantical definition of OCL constraints in the context of the MOF Model and its instances. The interpretation of these constraints on MOF compliant metamodels is realized by extending our validation framework through a MOF Bridge. Based on the MOF technology mapping for Java, it acts as a `ModelFacade` for MOF (s. chapter 3.4.1). An implementation of this Java mapping is eventually used to build a validator for OCL constraints on the MOF Model itself.

7.1 Accessing the MOF Model

Based on a traditional four layered metadata architecture, the MOF forms the foundation to describe the structure and semantics of metamodels. The layers of this architecture consist of the MOF Model at the M3, metamodels at the M2, model instances of these metamodels at the M1 and applications at the M0 level (s. figure 7.1).

The MOF Model provides a set of model elements classifying elements of a meta-

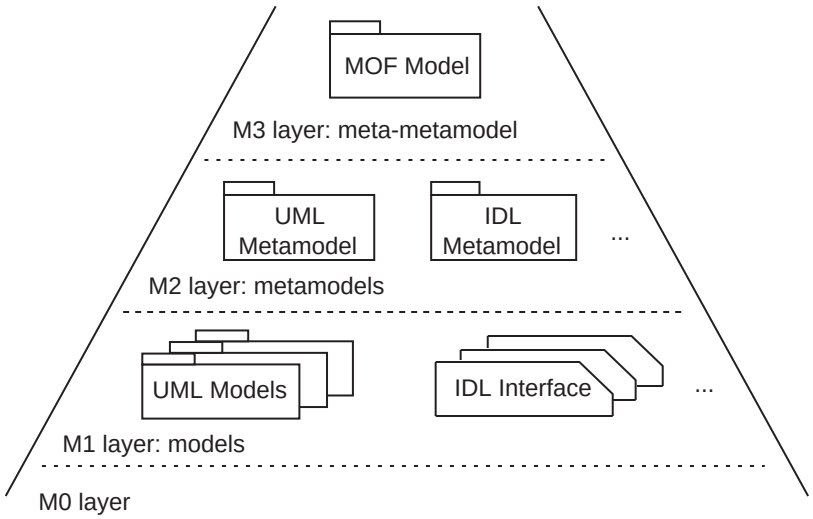


Figure 7.1: "A typical instantiation of the MOF metadata architecture with metamodels for representing UML diagrams and OMG IDL" [MOF02, p. 2-3].

model. By defining OCL expressions in terms of MOF Model elements, OCL constraints can be specified in the context of their metamodel's instances.

Required semantical and structural properties of these model elements and their instances are queried by using MOF technology mappings [MOF02, cha. 4].

7.2 The MOF-OCL Conceptual Binding

The basic assumption is that in general a MOF Model Element can be bound to an UML Model Element. We compare the individual model element's semantics by using their descriptions and constraints in form of relevant UML well-formedness rules and corresponding MOF Model Constraints respectively. This comparison leads us to limitations of a general binding which should be refined for individual metamodels.

7.2.1 Package

In the perspective of OCL a UML Package provides a way of partitioning and modularizing space of a model. Among others, it therefore contains other packages and classifiers. Analogously a MOF Package supports the containment of other packages (i.e. Package Nesting, Package Import and Package Clustering) and model elements (e.g. classifiers) at the M2 level. The relevant well-formedness rules of a UML Package in the context of OCL state, that names of contents of a package may not collide. This is also required for MOF Packages by the MOF Model Constraint [C-5] of a Namespace [MOF02, p. 3-90].

Additionally a MOF Package provides an *Extent* of the classes' instances at the M1 level [MOF02, cha 3.9.3.5], which can be used as codomain for the OCL property $\mathrm{allInstances}()$ (s. chapter 3.4.1).

By considering this, a MOF Package does not only represent the OCL concept of a package but also the set of instances each classifier defines.

7.2.2 Classifiers

Although a MOF Classifier defines a conceptual subset of an UML Classifier, the relevant OCL concepts are the same. That means that a classifier no matter how it is defined can be specified by its qualified name and is a container for a collection of features. The specification of classifiers and their features must be unambiguous in the context of OCL according to the well-formedness rules of a corresponding UML Classifier. We refer again to the MOF Model Constraint [C-5]. A MOF Classifier is contained by and is a MOF Namespace containing its features itself. Classifiers' qualified names and names of contained features can be resolved unambiguous by following the chain of namespaces.

However, the main difference between these two classifier definitions is that a MOF Classifier does not define an ownership to a state-machine. Therefore the UML concept of *States* as used in OCL cannot be represented in our binding. In case a MOF compliant metamodel specifies its own state, the general binding of a MOF Classifier to an UML Classifier must be refined within the respective concrete binding.

MOF Type	OCL Type
Package	Package
Classifier	Classifier
Enumeration Type	Enumeration
Boolean	Boolean
Integer	Integer
Long	Integer
Float	Real
Double	Real
String	String
Alias Type	same as of its base type
Collection Types	Classifier

Table 7.1: Corresponding MOF and OCL concepts

7.2.3 Enumeration Types

As a special classifier, a MOF Enumeration Type is related to an UML Enumeration. Although MOF enumerations are specified by labels instead of an UML Enumeration Literal's name, it is equivalent to the UML Enumeration. In the context of OCL it can be uniquely specified by its name. Therefore we can bind the MOF Enumeration Type to an UML Enumeration without loss of generality.

7.2.4 Primitive and Alias Types

The MOF primitive types are directly represented by their UML and OCL equivalents respectively. A detailed compilation can be found in Table 7.1.
In MOF the list of primitive types is supplemented by MOF Alias Types. These are types that are based on other data types, but "may convey a different "meaning" to that of its base type" [MOF02, p. 3-36]. An Alias Type is recursively bound to the corresponding concept of its base type.

7.2.5 Collection Types

Since there are no collection types in UML and the OCL inheritance hierarchy in relation to the OCL type OclAny does not allow a model type to subtype an OCL collection type, the MOF collection types must be handled as UML Classifiers instead of OCL collection types. Note that OCL collections containing MOF collections (and vice versa) are therefore not flattened [OCL03, cha. 6.5.13].

7.2.6 Features

MOF Operations, Attributes or Associations correspond to respective UML Operation, Attribute and Association model elements on the classifier and on the instance level.

Most of the well-formedness rules specify boundary conditions that are not relevant in this context. Feature names have to be unambiguous within a classifier. In MOF classifiers are special namespaces containing features. Again, the MOF Model Constraint [C-5] is applied for clearness.

Additionally, MOF Features allow to specify a multiplicity. This implies that attributes and operations can also have multiplicities greater or less than one. In both cases OCL does not define how to proceed, because the UML metamodel asserts that this value is a single value of a particular type. We therefore have to disregard the multiplicity specification in the general binding and delegate it to a validator implementation for a specific metamodel.

MOF Features/Associations	OCL Properties
Operation	Operation
Attribute	Attribute
Association	Association
Association End	Association End

Table 7.2: MOF Features and Associations in relation to UML/OCL concepts

UML Associations or Association Ends possess a multiplicity specification. Thus

MOF Associations are not constrained with regard to this concern. However, UML well-formedness rules define in contrast to the MOF Model, that an Association End must have a unique name within an Association. Though MOF does not demand this, its associations are defined to be binary. Starting at a particular association end, the navigation via the other association end's name is unambiguous. Thus MOF Associations and Association Ends can be bound without limitation to respective UML Associations and Association Ends.

This conceptual binding of MOF and UML is transitively used to bind MOF and OCL concepts (s. table 7.1 and 7.2). In general the following differences between MOF and UML cannot be generically resolved:

- MOF Packages additionally specify extents of M1 model instances
- MOF Classifier do not specify state-machines
- MOF Collection Types are not represented as collections
- MOF Attributes and Operations define multiplicities

7.3 Bridge Implementation Based on the JMI

In order to apply the MOF-OCL binding in our validator, we have to represent the metadata of a MOF compliant metamodel. This is done by using the Java mapping for MOF.

7.3.1 The Java Metadata Interface

The Java Metadata Interface (JMI) [JSR02] is a platform-specific MOF API for Java. It includes two groups of interfaces. The first group is related to the static structure of a metamodel. It describes the metamodel's elements and relations. The second group represents MOF reflection capabilities. These interfaces can be used to access a metamodel's model instance. Overall JMI allows us to discover the nature of metamodels on the M2 and M1 level independent from semantics.

7.3.2 Using the Bridge Design Pattern

The MOF concepts described by the JMI are accessed and subsequently classified to particular OCL concepts. The MOF Bridge therefore implements the validator's `ModelFacade` (s. chapter 3) by interpreting the obtained metadata of a described metamodel. Following its name, the MOF Bridge is implemented according to the *bridge* design pattern [GHJV95]. Thus individual elements of the conceptual bind-

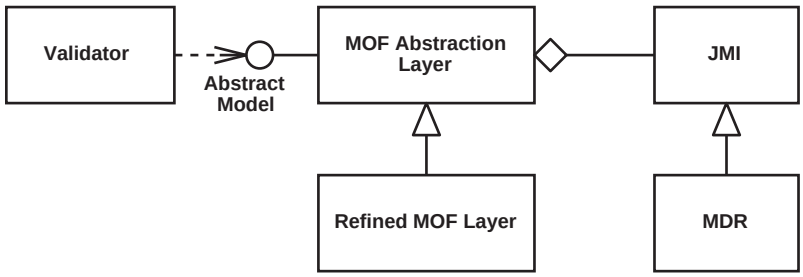


Figure 7.2: Simplified diagram of our MOF Bridge

ing and the JMI can be independently subclassed as depicted in figure 7.2. This decoupling enables refinement of the MOF-OCL binding and at the same time provides independence of JMI implementation and implementation technology.

7.3.3 Abstraction Layer Realization

In order to give an idea of the implementation of the MOF Bridge, typical access to a `ModelFacade` during the validation process is outlined. The MOF Bridge transforms requests and delegates them to JMI (s. figure 7.3). Requests on metamodels and instance models must be handled in different ways.

Requests on Metamodels The bridge uses the metamodel specific part of JMI composed by the so called `ModelPackage` to access the type level of OCL constraints. To obtain a specified package, its qualified name has to be resolved. Therefore the MOF Bridge applies the conceptual binding to transform the MOF Package

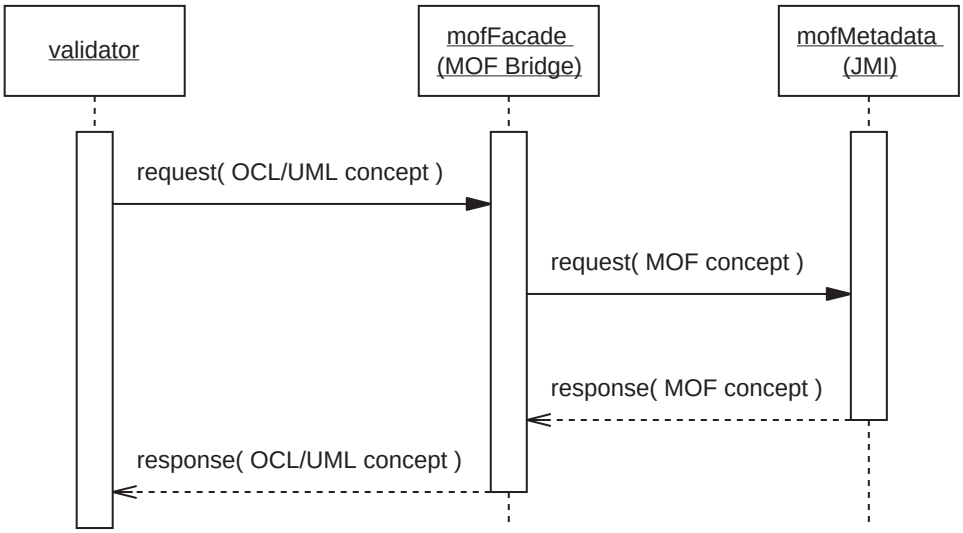


Figure 7.3: Sequence of MOF Bridge translations

representation `MofPackage` to a `Package` of our model interface and vice versa. This allows to access the package hierarchy of the metamodel from `ModelPackage`. By following the subsequent containers according to the specified name list, the package is resolved. The set of contained `Classifiers` is used to obtain OCL model types specified for instance in constraint contexts. If a classifier is found, the bridge will again translate the corresponding concepts. Contained properties and subsequent other concepts are queried in the same way. Overall, the whole type level is translated elementwise according to the presented binding and thus is accessible in the context of OCL.

Note that the MOF Bridge implementation uses a variant to represent association ends. Based on the analogous meaning of MOF References and Association with respect to navigation [MOF02, cha. 3.9.3.1], the implementation uses references instead of association ends for performance reasons.

Requests on Instance Models Scopes of OCL constraints are model classifiers and their instances. The instance level of a classifier is accessed within the evaluation process via the reflective module provided by the JMI.

In MOF, classifiers are subtypes of *RefObject* defining the reflective operation

refAllObjects(Boolean) [MOF02, cha. 6.2.3]. This operation returns the "the set of all Instances in the current extent whose type is given by this object's Class" [MOF02, 6-11].

Unfortunately this operation is not supported by JMI's reflective API. We circumvent this problem by accessing the metamodel's outermost reflective representation of a package `RefPackage` and use the method `refAllPackages()` to obtain the set of all packages. By calling `refAllClasses()` on these `RefPackages` we receive all classes contained in the whole metamodel. The searched instances can then be obtained with the operation `refAllOfClass()` returning all instances of particular classes and respective subtypes.

Features of instances can then be accessed and executed via the interface `RefFeatured`. The nature of resulting predefined parents (s. chapter 3.4.2) is obtained from JMI Java representations [JSR02, cha. 4.3].

For further information about the MOF Bridge and in particular its refinement, the reader is referred to the implementation and its API.

7.3.4 MDR-based MOF Model Support

Until now the MOF Bridge is defined and implemented independently of the choice of JMI technology. In order to use and represent specific metamodels for validators, the JMI compliant Meta Data Repository (MDR) [MDR03] is used.

MDR imports MOF instances described by XMI into its repository. Any MOF metamodel loaded into the MDR can be instantiated at runtime. The resulting JMI representation is used to exemplify the implementation of a concrete MOF Model OCL validator.

By importing the self-describing MOF Model using its XMI-description into the MDR we obtain its metadata in form of a JMI `ModelPackage` implementation. Furthermore a model instance must be imported in a specified `MOFPackage`. The resulting metamodel and instance extent represents sufficient information for our MOF Bridge and validator. In total this allows us to validate M2 level constraints such as MOF Model Constraints [MOF02, cha. 3.9] against M2 metamodels.

Summary

In this chapter a general binding describing the differences and similarities between the MOF and UML metamodel in terms of OCL has been developed. The resulting conceptual binding has been realized in the MOF Bridge. It translates metadata of M2 metamodels obtained from the JMI MOF Mapping to information required by our abstract model. This extension to our validator has been demonstrated by implementing the self-describing MOF Model using the MDR.

Chapter 8

Validation of UML Extensions

The UML metamodel is defined as an instance of the MOF Model. As such it is supported by our validator. This chapter uses the MOF Bridge to adapt the validation framework to the needs of this metamodel. Therefore the MOF Bridge's conceptual binding is refined. It is constructed using MDR metadata representing UML metamodel versions. This implementation enables validation of UML Profiles.

8.1 The UML Metamodel as Instance of the MOF Model

The UML metamodel in its varying versions is an instance of the MOF Model. Thus it is accessed by the validation framework using the MOF Bridge. The MOF Bridge uses a general conceptual binding that is refineable through its bridge implementation. This is used to adapt the MOF Bridge to the needs of the UML metamodel.

Chapter 7.1 indicates four issues subject to refinement: extents, collection types, states and multiplicity specifications of attributes and operations.

- The implementation limits the extent of a model instance to the UML metamodel package "UML".

- Collection types do not exist in the UML metamodel.
- Although the UML defines states the UML metamodel versions are stateless.
- The UML metamodel does not possess operations.

Thus we do not have to refine the MOF Bridge in respect of collection types, statemachines and multiple or non existing return values and arguments.

This, however, does not apply to attributes in UML. An example of a multivalued attribute can be found within the model element Stereotype. It defines the attribute *baseClass* representing a set of names. In order to support this kind of attributes we have to redefine the MOF Bridge.

An UML Attribute is an instance of a MOF Attribute. The MOF Bridge thus transitively binds an UML Attribute to an OCL Attribute according to the general conceptual binding (s. chapter 7.2). The general binding of MOF Attributes is set from attributes to associations. The refined validator thus handles attributes with multiplicity specifications unequal to one as associations. A multivalued attribute then returns a collection of instances, a singlevalued attribute an instance acting as collection and an undefined attribute an empty collection.

After defining the concrete binding of our MOF Bridge, we import the UML metamodel using its XMI representation into the MDR MOF Repository. As a placeholder for current and future UML metamodel versions, we use the resulting metadata to discover the structure and semantics of the most commonly used UML Metamodels – the versions 1.3 and 1.4. This implementation checks OCL M1 level constraints like UML well-formedness rules against respective model instances.

8.2 Validation of UML Profiles

A Profile is a specialized package containing a set of UML Stereotypes, Tag Definitions and Constraints located in the extension system of the UML. Using stereotypes and tagged values the semantics of a model element of the UML metamodel can be

narrowed to the requirements of a particular domain. The semantics of such specialized model elements are defined by constraints. The subset of OCL constraints can be extracted to a file conforming to the OCL grammar. The framework extension for UML checks and evaluates this file in the context of the metamodel and a model instance. Thus this framework implementation validates OCL constraints for the UML Profiles extension mechanism.

Summary

This chapter has refined the MOF Bridge to the needs of the UML metamodel by handling non single attributes as associations. The resulting interpreter validates OCL Constraints defining semantics of UML Profiles for domain specific model instances.

Chapter 9

Conclusion

As introduced in chapter 1, a validator for UML Extensions has been developed and implemented. Therefore OCL has been adjusted with respect to inconsistencies and inaccuracies. A model interface has been developed describing arbitrary model elements in terms of UML model elements. This description has been transitively used to bind respective model elements to concepts of OCL. The interface has allowed to define and interpret OCL constraints on arbitrary models.

Based on this interface, an extensible interpreter framework has been developed. It chains a parser, a context checker and an interpreter. To use common parser and compiler generator technologies, the original OCL grammar has been adjusted to an LALR(1)-parseable grammar. Based on the SableCC framework, a corresponding parser has been generated. Context dependent OCL features are checked on produces ASTs in the context check. A type system abstraction has been developed to uniformly represent types originated from the model or predefined in OCL. The type system have been made extensible by describing predefined types via the Java Reflection API. During the type check resolved types has been stored in an environment. The interpreter reuses these types in the evaluation process of constraints. Analogous to the type check an evaluation algorithm has been defined. The model interface has been extended to access the results of evaluated constraints for validation purposes.

The abstraction layer of this interpreter framework has then been implemented with semantics of MOF model elements obtained from the JMI. The resulting MOF Bridge

extends the framework to support OCL constraints defined on MOF compliant metamodels. The bridge has been refined to the needs of the UML metamodel. This allows to support this metamodel in its varying versions. The MDR MOF repository has been used as an implementation of the JMI. By importing corresponding XMI representations into the MDR, a validator for the MOF Model and the UML metamodel version 1.3 and 1.4 has been realized. The latter subsequently supports UML metamodel extensions in form of UML Profiles defining OCL constraints.

Outlook

While this thesis has been developed standards have progressed. The UML specification and in parallel the MOF standard are advancing to version 2.0. The UML and MOF 2.0 infrastructure are developed to share a common core. Analogously to the binding mechanism developed in this thesis, the resulting similarities provide the means to use a subset of OCL on MOF compliant metamodels. As of now, the sixth revision of OCL 2.0 does not specify the exact extent of related OCL concepts [ocl04, cha. 14]. By implementing the final relation in the model interface, this framework can be used to test and support the subset of OCL features in the boundaries of OCL 1.5. Depending on this extent an OCL 2.0 validator should match the capabilities of this framework with respect to MOF compliant metamodels.

> "Change does not necessarily assure progress, but progress implacably requires change. Education is essential to change, for education creates both new wants and the ability to satisfy them."
> – Henry Steele Commagert.

Appendix A

SableCC Grammar

In SableCC productions do not allow the use of parenthesis. The offending grammar productions are *unfolded* to avoid this problem. The resulting grammar expansion does not change the language itself [Pep97]. This grammar complies to the changed grammar introduced in our thesis and therefore possess its addressed characteristics.

```
Package de.tuberlin.cs.cis.ocl.parser;

Helpers

    unicode_character = [0..0xffff];
    ascii_letter =  ['A' .. 'Z'] | ['a' .. 'z'];

    digit = ['0'..'9'];

    ht = 0x0009;    // tab
    lf = 0x000a;    // linefeed
    ff = 0x000c;    // formfeed
    cr = 0x000d;    // carriage return
    sp = ' ';       // space

    line_terminator = lf | cr | cr lf;
```

```
    /* name: String - "To minimize portability problems use names
     * that start with ASCII letter, and consist of ASCII letters
     * and digits, space and underscore." [MOF02, p. 3-16]
     * Seems rational, done! But spaces won't be allowed for
     * parsing reasons.
     */
    char_for_name_top = ascii_letter | '_';
    char_for_name =  char_for_name_top | digit;

    /* This helper is actually not needed because of our
     * definition of char_for_name and char_for_name_top helper,
     * but for the sake of completness as helper function.
     */
    inhibited_chars =   ' ' | '"' | '#' | ''' |'(' | ')' |
                        '*' | '+' | ',' | '-' | '.' | '/' |
                        ':' | ';' | '<' | '=' | '>' | '@' |
                        '[' | '\' | ']' | '{' | '|' | '}';


    input_character = [unicode_character - [cr + lf]];
```

Tokens

```
    /* For a given input, the longest matching token will be
     * returned by the lexer. In the case of two matches of the
     * same length, the token listed first in the specification
     * file will be returned. Thus identifier should be the last
     * token in order to avoid failures.
     */

    white_space = (sp | ht | ff | line_terminator)*;

    end_of_line_comment = '--' input_character* line_terminator?;
```

```
    set = 'Set';
    bag = 'Bag';
    sequence = 'Sequence';
    collection = 'Collection';

    dot = '.';
    arrow = '->';

    not = 'not';

    mult = '*';
    div = '/';
    plus = '+';
    minus = '-';

    context = 'context';

    pre = 'pre';
    post = 'post';
    inv = 'inv';
    def = 'def';

    equal = '=';
    n_equal = '<>';
    lt = '<';
    gt = '>';
    lteq = '<=';
    gteq = '>=';

    and = 'and';
    or = 'or';
    xor = 'xor';
    implies = 'implies';
```

```
    l_par = '(';
    r_par = ')';
    l_bracket = '[';
    r_bracket = ']';
    l_brace = '{';
    r_brace = '}';
    semicolon = ';';

    dcolon = '::';
    colon = ':';
    comma = ',';
    at = '@';
    bar = '|';
    ddot = '..';

    if = 'if';
    then = 'then';
    else = 'else';
    endif = 'endif';

    boolean_literal = 'true' | 'false';

    let = 'let';
    in = 'in';

    package = 'package';
    endpackage = 'endpackage';

    number_literal =
        digit (digit)*                                  // integer
        ( '.' digit (digit)*)?                          // decimal
        ( ('e' | 'E') ('+' | '-')? digit (digit)* )?    // exponent
    ;
```

```
    string_literal =
        '''
        ([[[unicode_character - [cr + lf]] - '''] - '\']
         | '\'
            (
                // escape sequences
                'n' | 't' | 'b' | 'r' | 'f' | '\' | ''' | '\"'

                // octal escape
                 | (['0'..'7'] ['0'..'7'] ['0'..'7']?)?)
            )
        ) *
        '''
    ;

    identifier = char_for_name_top char_for_name*;

Ignored Tokens

    white_space, end_of_line_comment;

Productions

/************************************************************
        file structure and constraint declarations
************************************************************/

    ocl_file = ocl_package+;
        ocl_package = package package_name constraint* endpackage;

    package_name = path_name;
```

```
// changed: replaces 'oclExpressions'
// kleenex operator relocated to 'oclFile'.
constraint = context_declaration context_bodypart+;
    context_bodypart =
        {definition}    def name? colon let_expression* |
        {constraint}    stereotype name? colon ocl_expression;

context_declaration = context context_kind;
    context_kind =
        {operation} name dcolon context_operation_name l_par
           formal_parameter_list r_par return_type? |
        {classifier} name classifier_type?;
            return_type = colon type_specifier;
            classifier_type = colon name;

stereotype =
    {pre_condition}     pre |
    {post_condition}    post |
    {invariant}         inv;

// changed: re-use of the predefined operator-productions (3)
context_operation_name =
    name |
    {logical}       logical_operator |
    {relational}    relational_operator |
    {add}               add_operator |
    {multiply}      multiply_operator;

formal_parameter_list = param_list?;
    param_list = formal_parameter next_param*;
        formal_parameter = name type_postfix;
        next_param = comma formal_parameter;
```

```
/************************************************************
        expressions
************************************************************/

    ocl_expression = let_declaration? expression;
        let_declaration = let_expression* in;

    let_expression = let name let_parameter_list? type_postfix?
       equal expression;
        let_parameter_list = l_par formal_parameter_list r_par;
        type_postfix = colon type_specifier;

    if_expression =
        if [condition] : expression
        then [then_branch] : expression
        else [else_branch] : expression endif ;

    expression = logical_expression;

    // changed: boolean operators have higher precedence than the
    // implies operator (8)
    logical_expression = boolean_expression implication*;
        implication = implies_operator boolean_expression;

    // added (8)
    boolean_expression = relational_expression boolean_operation*;
        boolean_operation = boolean_operator relational_expression;

    // changed: compare operators have higher precedence than
    // the '=' and '<>' operators(9)
    relational_expression = compareable_expression equation?;
        equation = equation_operator compareable_expression;
```

```
// added (9)
compareable_expression = additive_expression comparison?;
    comparison = compare_operator additive_expression;

additive_expression = multiplicative_expression addition*;
    addition = add_operator multiplicative_expression;

multiplicative_expression = unary_expression multiplication*;
    multiplication = multiply_operator unary_expression;

unary_expression =
    {prefixed}  unary_operator postfix_expression |
    postfix_expression;

postfix_expression = primary_expression property_invocation*;
    property_invocation =
        {object}        dot property_call |
        {collection}    arrow property_call;

primary_expression =
    {collection}    literal_collection |
    {literal}       literal |
    {property_call}     property_call |
    {parenthesed}       l_par expression r_par |
    {if}            if_expression;

time_expression = at pre;
```

```
/************************************************************
        property calls
************************************************************/

    property_call = path_name time_expression?
        qualifiers? property_call_parameters?;

    property_call_parameters =
        l_par declarator? actual_parameter_list? r_par ;

    actual_parameter_list = expression next_expr*;
        next_expr = comma expression;

    // changed: production not LALR(1)-parsable (5)
    // workaround: names wrapped by expressions
    declarator =
        {concrete} actual_parameter_list simple_type_postfix?
           accumulator? bar |
        (name_list simple_type_postfix? accumulator? bar);
            accumulator = semicolon name colon type_specifier equal
                expression;
            name_list = name next_name*;
            next_name = comma name;
            simple_type_postfix = colon simple_type_specifier;

    qualifiers = l_bracket actual_parameter_list r_bracket;

/************************************************************
        operators
************************************************************/

    // changed: seperation of implies and boolean operators (8)
    logical_operator = boolean_operator |
        {implicative} implies_operator;
```

```
// added (8)
boolean_operator =
    {and}     and |
    {or}      or |
    {xor}     xor;

// added (8)
implies_operator = implies;

relational_operator =
    {equality}  equation_operator |
    {compare}   compare_operator;

equation_operator = equal |
    {in}    n_equal;

compare_operator =
    {gt}      gt |
    {lt}      lt |
    {gteq}    gteq |
    {lteq}    lteq;

add_operator =
    {plus}  plus |
    {minus} minus;

multiply_operator =
    {mult}  mult |
    {div}   div;

unary_operator =
    {minus} minus |
    {not}   not;
```

```
/************************************************************
        literals and type specifiers
************************************************************/

    type_specifier =
        {ocl_any}       simple_type_specifier |
        {collection}    collection_type;

    simple_type_specifier = path_name;

    collection_type =
        collection_kind l_par simple_type_specifier r_par;

    collection_kind =
        {set}       set |
        {bag}       bag |
        {sequence}  sequence |
        collection;

    literal_collection =
        collection_kind l_brace collection_item_list r_brace;
        collection_item_list = collection_item next_collection_item
           *;
            next_collection_item = comma collection_item;

    collection_item = expression range?;
        range = ddot expression;

    // changed: enum_literal is left out (6)
    // added: boolean literal (7)
    literal =
        {string}    string_literal |
        {number}    number_literal |
        {boolean}   boolean_literal;
```

```
/***********************************************************
        names
***********************************************************/

    name = identifier;

    path_name = name_qualifier* name;
        name_qualifier = name dcolon;

    /* removed: united with name (1)
    type_name = identifier */
```

Appendix B

Zusammenfassung

Die Object Constraint Language (OCL) ist eine formale Sprache, die es erlaubt, Semantiken in Form von Beschränkungen und Erweiterungen für Modelle und Metamodelle zu spezifizieren.

Um die Einhaltung dieser Semantiken überprüfen zu können, wird in dieser Diplomarbeit ein OCL-Interpreter entwickelt und implementiert. Dieser hat die Aufgabe, auf der Metamodell-Ebene spezifizierte OCL Constraints der Version 1.5 zu validieren.

Hierfür wird, der klassischen *pipe-and-filter* Architektur folgend, ein Parser, ein Context Checker und ein Interpreter realisiert. Diese beruhen auf einer abstrakten Modellbeschreibungsschnittstelle, die es erlaubt, Metadaten eines objektorientierten Modells im Kontext der OCL zu interpretieren. Diese Beschreibung erweiternd wird eine Brücke auf Basis des Java Metadata Interfaces entwickelt, die es erlaubt, Semantiken beliebiger MOF konformer Metamodelle zu validieren.

Als Instanzen des MOF Modells ist es mit diesem Parser-Framework abschliessend möglich, die verschiedenen Versionen des UML Metamodells in Form von OCL-Semantiken und Erweiterungen zu interpretieren und zu validieren.

Bibliography

[API] Sun Microsystems, Inc. *Requirements for Writing Java API Specifications*. `http://java.sun.com/j2se/javadoc/writingapispecs/index.html`.

[ASU99] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullmann. *Compilerbau Teil 1*. Oldenbourg, 1999.

[Con03] The Unicode Consortium. *The Unicode Standard, Version 4.0.0, defined by: The Unicode Standard, Version 4.0*. Boston, MA, Addison-Wesley, 2003. ISBN 0-321-18578-1.

[Fin00] Frank Finger. Design and Implementation of a Modular OCL Compiler. Master's thesis, Dresden University of Technology, 2000.

[Gag98] Etienne Gagnon. SableCC, an Object-Oriented Compiler Framework. Master's thesis, McGill University, Montreal, March 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Java Series. Sun Microsystems, 1996.

[IBM03] THE IBM OCL Parser. Website, 2003. `http://www.software.ibm.com/ad/ocl`.

[ISS03] UML 1.4 RTF : OCL Issues. December 2003. `http://www.klasse.nl/ocl/ocl-issues.pdf`.

[j2s03] Java™2 Platform, Standard Edition, v 1.4.2 API Specification, 2003. `http://java.sun.com/j2se/1.4.2/docs/api/`.

[JCC03] Sun Microsystems, Inc. *JavaCC [tm]: Grammar Files*, 2003. `https://javacc.dev.java.net/`.

[JSR01] JSR-014, Adding Generics to the Java™Programming Language. Website, 2001. `http://www.jcp.org/aboutJava/communityprocess/review/jsr014/`.

[JSR02] JSR-040, The Java™Metadata Interface (JMI) Specification. Website, 2002. `http://jcp.org/aboutJava/communityprocess/final/jsr040/index.html`.

[MDR03] Metadata Repository (MDR). Website, 2003. `http://mdr.netbeans.org/`.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, second edition, 1997.

[Mic97] SUN Microsystems. Java core reflection api and specification. Technical report, SUN Microsystems, 1997.

[MOF02] Object Management Group (OMG). *Meta Object Facility (MOF) Specification*, 2002.

[OCL03] *Object Management Group (OMG). Object Constraint Language [UML03, 6-1]*, 2003.

[ocl04] Unified modeling language: Ocl version 2.0, 2004. `http://www.omg.org/docs/ptc/03-08-08.pdf`.

[Pep97] Peter Pepper. Programmiersprachen und -systeme. 1997.

[Pep99] Peter Pepper. *Funktionale Programmierung in OPAL, ML, HASKELL und GOFER*. Springer, 1999.

[RM-95] International Organization for Standarization. *Basic Reference Model of Open Distributed Processing*, 1995.

[UML03] Object Management Group (OMG). *Unified Modeling Lanuguage Specification 1.5*, 2003. `http://cgi.omg.org/docs/formal/01-09-67.pdf`.

[WA99] Jos Warmer and Kleppe Anneke. *The Object Constraint Language : Precise Modeling with UML*. Addison-Wesley, 1999.

[XMI03] Object Management Group (OMG). *XML Metadata Interchange (XMI)*, 2003. `http://www.omg.org/technology/documents/formal/xmi.htm`.