«Java»OCL Based on New Presentation of the OCL-Syntax

Bernhard Rumpe

Technische Universität München,
Arcisstr. 21, D-80333 Munich, Germany,
Rumpe@in.tum.de

Abstract The Object Constraint Language (OCL) is a part of the Unified Modeling Language (UML) – an emerging standard language for object-oriented analysis and design. OCL is designed as a formal language for specifying constraints that cannot be expressed conveniently using UML's diagrammatic notation.
This article describes results of a careful analysis of the syntactic structure of OCL, resulting in a number of improvements of the OCL-syntax. In particular, a new and better readable grammar describing OCL is defined. The paper enhances not only the language OCL itself, but in particular its presentation.
Given the new grammar, a Java-style variant of OCL with essentially the same abstract grammar is defined, which should be more comfortable to Java-programmers.

1 Introduction

The Unified Modeling Language (UML) [17] has become the de-facto standard for modeling object-oriented systems. The UML is a graphical description language and therefore, similar to most other graphical languages, limited in its expressiveness. The Object Constraint Language (OCL) [16] is a precise textual specification language used to complement graphical modeling languages. OCL was designed to express logical constraints within a UML model that cannot or not conveniently enough be expressed with the mainly diagrammatic UML. In particular, OCL supports specifying invariants of classes in a class model and describing pre- and postconditions of operations and methods.

OCL is a textual supplement of UML. As such, it needs to be pragmatic in use, but still concise enough in its definition to be assisted by tools, such as a parser and a type or consistency checker.

The history of language definitions shows, that it is impossible to define a sufficiently useful language in perfect shape from the very beginning. Many different syntactic, semantic, and methodical issues have to be resolved, and different stake-holders want a language adapted for particular needs. Therefore, it is not surprising that the relatively young OCL has some syntactic and semantic flaws that need to be fixed. Fortunately, OCL can build on a large basis of work on already defined languages, starting from the programming languages

[Rum02b] B. Rumpe.
Java-OCL Based on New Presentation of the OCL-Syntax.
In: Object Modeling with the OCL, pp. 189-212,
Tony Clark, Jos Warmer (eds.)
LNCS 2263, Springer Verlag, 2002.
www.se-rwth.de/publications

like Modula-2 [21] or Java [9], including OCL's own predecessor [6], up to textual specification languages such as VDM [7] and Spectrum [3]. The research on language definition shows that not only the language itself, but also its presentation can be improved to make it more amenable for tools as well as for reading and understanding the language.

The purpose of this article is not to provide OCL with semantics. Instead this article concentrates on the syntax and the presentation of this syntax of OCL. In Section 2 separation between syntax and its presentation is discussed in detail. Section 3 furthermore discusses how to use OCL at the meta-level without explicitly including the meta-level of OCL. In Sections 4 and 5 the original OCL grammar given in the OCL specification is re-formulated and the context conditions are adapted based on the insights gained before. Section 6 then presents a syntactic variant of OCL that is oriented towards Java. It is called ≪Java≫OCL and might be regarded as Java-profile of OCL.

It is assumed that the reader is somewhat familiar with OCL as well as with UML class diagrams.

2 Langauges and their presentation

This paper restructures the presentation of the expression part of OCL so that its grammar rules are structurued in a way similar to that of Java (and C++). To understand the impact of such a restructuring, it is necessary to look at how language definition works in general. Formally, a textual *language* is a set of *well-formed sentences* over a basic alphabet. This holds for natural languages as well as for Java and OCL.

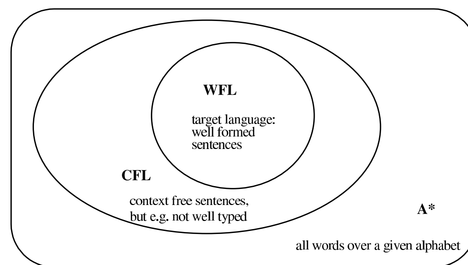

Figure 1. The hierarchy of language definition

Given an alphabet `A` (e.g. the ASCII or Unicode character sets), a language is a subset of all words over that alphabet. Among others, Java, Pascal and OCL are examples. Due to the fact that these languages have infinite numbers of sentences (i.e. possible class definitions in Java), a finite, compact and understandable characterization of such a language is necessary.

Normally a compact definition is achieved in two steps. In a first step the so called *context free language* is defined using a grammar. The grammar of a language therefore *presents* a language. The extended Backus-Naur-Form (EBNF) is a comfortable way to describe the grammar. Its conventions are[1]:

- Identifiers included in ⟨ ⟩ brackets denote nonterminal symbols.
- **Underlined boldface** font denote terminal symbols.
- The empty word is denoted by ε.
- Brackets {...} are used for grouping.
- Alternatives are separated by a vertical bar |.
- Constructs followed by a Kleene star, such as $\langle\text{identifier}\rangle^*$ and $\{\ldots\}^*$ can be repeated zero or more times.
- Optional parts are expressed by a question mark $^?$.

The context free grammar of a language defines the set of context free sentences, called `CFL`. However, still many ill formed OCL-expressions may exist. Well known examples are violated typing rules or variables that are used, but not declared or have a wrong type. Therefore, in a second step, context conditions further constrain the `CFL`, resulting in a set of well-formed sentences (`WFL`). Standard examples for context conditions are typing rules as well as rules for variables declarations and uses. Unfortunately, there is no simple technique similar to a context free grammar to describe context conditions and therefore these context conditions are usually presented as informal, textual description together with examples. In summary, there are three sets of sentences (see Fig. 1) including each other: $\texttt{WFL} \subset \texttt{CFL} \subset \texttt{A}^*$. In the context of OCL this means, the set of well formed, useful OCL expressions is identified with `WFL`.

The following sections concentrate on the context free language definition of OCL (called `CFL`) and the improvement and reformulation of the context conditions is left to others (see related publications in this proceedings).

Context conditions are often referred to as *semantic conditions*. This is partially a historic fault, because context conditions are the last step to define the language. They are heavily influenced by the intended language semantics, but nevertheless they are defined in form of purely syntactic and therefore checkable criteria. In particular, context conditions do not define the semantics of a language. E.g. from a given set of context conditions on a language like OCL or C++ the semantics of the `and`-operator on undefined values or non-terminating calculations cannot be inferred. The reader is referred to [12] for a general discussion of semantics for modeling languages such as the OCL.

Furthermore, the separation between context free grammar and context conditions is floating. Although, context conditions exist that can clearly not be expressed in a context free grammar, there are other issues that can be defined in context free grammars, but usually aren't. Among them are the priorities of infix operators. The OCL book defines them in context free form, but gets blurred through a number of additional nonterminals (⟨expression⟩,

[1] EBNF provides further operators that serve as notational shortcuts, but aren't used here.

⟨ifExpression⟩, ⟨logicalExpression⟩, ⟨relationalExpression⟩, ⟨additiveExpression⟩, etc.). It's more compact, readable and equally precise to use an explicit precedence table, like the one in Table 1 that expresses the same information in a more compact way. Therefore, the relationship between `WFL` and `CFL` of a language can strongly be influenced by an intelligible definition.

Name	Syntax	Associativity
Unary operations	`-, not`	right
Multiplication and division	`*, /`	left
Addition and subtraction	`+, -`	left
Relational operations	`<, >, <=, >=`	left
Relational equality	`=, <>`	left
Logical operations	`and, or, xor`	left
Logical implies	`implies`	right

Table 1. Precedence for OCL operations (highest to lowest).

Furthermore, a distinction between the grammar that characterizes a language and the language itself is important. There are numerous grammars that describe the very same (context free) language. As a simple example, the grammar $\langle X\rangle ::= \underline{\mathbf{a}}\ \{\ \underline{\mathbf{b}}\ \langle X\rangle\ \underline{\mathbf{a}}\ \}^*\ \underline{\mathbf{b}}$ describes alternating `ab`-structures. Another grammar for the very same "language" can be defined by:

$$\begin{aligned} \langle \mathtt{Y}\rangle \ &\mathtt{::=}\ \langle \mathtt{ADAM}\rangle^* \ \langle \mathtt{ADAM}\rangle \ \langle \mathtt{ADAM}\rangle^* \\ \langle \mathtt{ADAM}\rangle \ &\mathtt{::=}\ \underline{\mathbf{a}}\ \langle \mathtt{EVE}\rangle \\ \langle \mathtt{EVE}\rangle \ &\mathtt{::=}\ \underline{\mathbf{b}}\ \langle \mathtt{ADAM}\rangle^? \end{aligned}$$

Both grammars present the same language $\underline{\mathbf{ab}}\{\underline{\mathbf{ab}}\}^*$, but strongly differ in their structure. One important purpose of this article is to restructure the given context free grammar of OCL. These changes aim at an improvement of the grammar of OCL to become more amenable to an interested reader, as well as to be more compact for a tool implementation. Along the way of improving the grammar, there will also be a number of language improvements that will be discussed in the next sections.

3 The use of OCL. What are meta-levels good for?

3.1 Fitting OCL into a language category

The Object Constraint Language (OCL) defines itself as "an expression language that enables one to describe constraints on object-oriented models and other modeling artifacts." ([20], pg. xix). Heavily based on the *constraint* concept, it explicitly gives its own interpretation of this term in an object-oriented setting by "a constraint is a restriction on one or more values of (part of) an object-oriented model or system." ([20], pg. 1).

However, OCL exhibits some characteristics of other kinds of languages and lacks some standard concepts of constraint languages. Therefore, a short comparison of OCL to other languages seems appropriate. The main language categories that are not necessarily disjoint are:

modeling language allows abstract characterizations of the modeling domain, which may be a system as well as business or other forms of environment.
object-oriented language encompasses the concept of *object*, together with dynamic creation, identity, and normally also inheritance and classes.
programming language is primarily characterized through being executable in such a way that it constructively transforms its input (and internal data structures) into an output result.
logic language provides logic concepts, such as Boolean operators and quantifiers to describe properties of systems and their data structures.
functional language is an highly compact, executable language that operates without side-effects.

OCL exhibits some characteristics of most of these languages. However, it is not a full modeling language, because OCL does not provide concepts to define new data structures. New classes, methods, or constants cannot be introduced. Therefore, OCL is only intended to be used together with an existing modeling language, such as UML, that needs a supplement for describing constraints. In this combination OCL also exhibits the characteristics of an object-oriented language, because it incorporates the UML typing system.

The relationship of OCL to logic is somewhat difficult. Yes, OCL provides Boolean operators, like `and`, etc., but these are present in programming languages, like Java, as well. The basic question here is, how OCL treats undefined values. To show the differences, Fig. 2 gives an overview of possible semantics of the `and`-operator, where some are useful for logic and some are useful for execution. Differences in the presented truth tables are underlined. E.g. in C++ `A and B` and its commutative `B and A` are distinguished, where in logic languages this hampers the usability of a language. Current definitions of OCL take different and sometimes inconsistent approaches [20].

Yes, OCL provides two quantifiers, `forall` and `exists`, but only for finite domains. Therefore, OCL is only a propositional logic. This means that without further concepts, OCL is neither capable of specifying the transitive closure of the subclass relation ([5]) nor many other recursive or loop based properties. Fortunately, OCL provides some built in data types such as Integer or Strings and can rely on UML based data types as well. Based on given method implementations of the UML model it is possible to specify many constraints beyond propositional logic. In summary, OCL is a propositional specification language that exhibits additional power through data structures and operations provided by UML.

OCL is not a general programming language. It is widely believed, that OCL is executable. This belief is correct in the sense that there are algorithms that can decide on constraints written in propositional logic. However, there are a number of subtle problems among others with non-terminating operations provided by

Classic 2-valued Logic:

$a \wedge b$	True	False
True	True	False
False	False	False

Strict, implementation:

a and b	True	False	Undef
True	True	False	Undef
False	False	False	Undef
Undef	Undef	Undef	Undef

Parallel implementation and Kleene-Logic:

$a \wedge b$	True	False	Undef
True	True	False	Undef
False	False	False	False
Undef	Undef	False	Undef

Sequential, for implementation (Java):

a && b	True	False	Undef
True	True	False	Undef
False	False	False	False
Undef	Undef	Undef	Undef

Lifting (Undef treated as False):

$a \wedge b$	True	False	Undef
True	True	False	False
False	False	False	False
Undef	False	False	False

Figure 2. Possible implementations of the `and`-operator

UML. Assuming that OCL is executable, it unfortunately cannot constructively change any data structure, because it can only compute Boolean values that tell, whether some constraint is violated or not. E.g. if a post-condition of form `a<b` is given, then it can check whether that condition holds. However, it cannot easily correct that condition. It may increase `b` or it alter `a`, or both? Although, there are standard solutions for particular cases, e.g. there cannot be a solution for general constraints. For example in `f(a)=g(b)` there is no general solution if it is assumed that both methods are provided by underlying UML models and OCL doesn't know anything about them.

Under the assumption that OCL is executable, it is very similar to the expression part of a functional language. It calculates values without changing the underlying data structures and therefore without side-effects. If an executable version of OCL was embedded in an ordinary language, e.g. by using OCL as expression sublanguage then a compact general programming language would be given. GOS [19] is an example for such a successful combination.

3.2 Where to use OCL?

Being a mixture of propositional logic and functional language that heavily depends on the underlying UML models OCL can serve a number of different purposes.

OCL code may be added to a running system for testing purposes. The OCL constraints are then used like assertions in C++ or Eiffel's contracts in form of pre- and postconditions.

This may be a very effective means of defining model based tests. If direct generation of code from OCL is feasible and this is ideally accompanied by code generation from the UML model itself, then the *Extreme Modeling* approach [2,14] becomes feasible. The basic idea of the Extreme Modeling approach is the

replacement of an ordinary programming language for coding and test specification by a high-level graphical approach, based on UML and OCL. OCL then plays a vital role in the software development process as a constraint definition language for the description of automated tests. This can greatly enhance the productivity of the programmers. Furthermore, modeling and programming become one combined activity.

It is worthwhile to distinguish development-time and runtime. During *development-time* the system is being developed. The source code and models of the system are available and these artifacts are iteratively enhanced, detailed and transformed. During *runtime* the source code has been translated into object code and is executed. Above the usefulness of OCL during runtime was discussed. The benefits of OCL during development-time and which OCL constraints can be executed during development is discussed below.

For stability of the running application, for efficiency and a number of other reasons, it makes sense to restrict reflective access and modifiability of systems during runtime. Although there are systems that allow dynamic adaptation, and this is sometimes very useful, in most cases this kind of runtime adaptability is rather restricted and deals with coarse modifications only. E.g. Java allows dynamic load of complete classes or packages with well defined interfaces, but not to adapt single methods or attributes within a class. Common Smalltalk systems of course are exceptions to that rule, because they allow rather unrestricted access to their own meta-levels. However, this feature should not be used widely as it is dedicated to experienced programmers mainly. Further drawbacks of reflective access are the increased complexity for understanding the code and the lack of a typing system.

However, reflection in sequential programming languages is generally understood by now. In a specification language, such as OCL, the access of the meta-level directly within OCL is widely unclear and unnecessary. First, it is widely unclear, because it allows to specify weird statements, such as class `Car` does exist exactly, if the attribute `age` of object `tim:Person` is not 5:

```
tim.age <> 5  xor  OclType.allInstances( x | x.name <> "Car" )
```

How is that to be implemented? a) ensure `tim` is older than 5, b) ensure `tim.age=5` and don't implement class `Person`, or c) delete class `Person` when `tim` becomes 5?

`tim.age <> 5` is an expression to be executed during runtime, the other part is to be evaluated during development-time where objects like `tim` aren't yet available. OCL, therefore, mixes evaluation times by providing direct access on the meta-level.

The primary purpose of the OCL meta-level is to allow the developer to talk about properties of a UML (and OCL) model during development-time, such as: "If class `Person` provides method `foo()`, then so does class `Child`". Fortunately meta-level access is not necessary within OCL itself. The trick is to provide this meta-level access through the underlying UML meta-model. `OclType` then becomes an ordinary UML class and is not part of OCL itself.

Furthermore, a specification language for describing runtime behavior e.g. in form of pre- and postconditions and another specification language that describes restrictions on the modeling elements must be strictly distinguished. The idea is to use OCL at both levels and to call them *meta-OCL* and *runtime-OCL*. Both are entirely disconnected and just by coincidence have the same look-a-like. The English language e.g. can be described using the English language itself, but it could also be described using an entire different language. Many other examples, where languages are used to describe themselves can be found. E.g. the EBNF to describe context free grammars can itself be defined using EBNF – both appearances of EBNF are fully separated.

Runtime-OCL talks about objects and values. The term `tim.age <> 5` belongs to this language. Meta-OCL talks about classes, their attribute structure and associations. The term `OclType.allInstances(x|x.name<>"Car")`, which can easily be checked during development-time, would belong to Meta-OCL. A combination of both expressions would be syntactically disallowed. OCL would be type checked and executable at both levels separately.

At development-time OCL can describe constraints that may not be violated by UML models. E.g. if a class is tagged by a certain stereotype «`EJB`», then it should provide certain functionality. OCL is surely feasible here. However, it would be of interest to link meta-OCL constraints with procedures at the meta-level that help to repair violated constraints, e.g. through automatically or interactively adding required functionality. Meta-level procedures act during development-time, where the developer is still available.

In summary, it becomes clear that a meta-level needs not be accessible within OCL. As a conclusion the meta-level of OCL is removed in the following. The OCL grammar is in the following restructured and the OCL context conditions are re-formulated appropriately, so that this flaw in the presentation of OCL can be amended. Furthermore, OCL becomes much simpler and more understandable to programmers.

4 The new OCL Grammar

OCL can be described by a context free grammar like any other language. In this section, an enhanced version of such a grammar through refactoring of the grammar originally given in [16] will be provided.

One of the most important changes concerns the strict separation of logic level and meta-level discussed in Section 2. In specification languages, e.g. SPECTRUM [3] and in all strongly typed programming languages, meta-level and logic level are never mixed.

The redefined grammar below is partly supplied with an explanation where considered necessary. Please note that is is assumed that the reader has some familiarity with OCL as provided in [16]. The following grammar rules are all formulated in EBNF as introduced in the previous section.

In the grammar's restructuring the identifiers ⟨className⟩, ⟨attributeName⟩, ⟨roleName⟩, ⟨methodName⟩, ⟨packageName⟩, ⟨stateName⟩, and ⟨varName⟩ were

introduced supplementary. They are equal to ⟨name⟩. Using these identifiers instead of ⟨name⟩ makes the grammar easier to read.

Note that the grammar restructuring to a large extent focuses on a better grammar presentation, not on the language adaptation, even if some changes are made. The following section 5.4 discusses these changes on the language itself.

A constraint can either specify an invariant or describe pre- and/or postconditions but never do both at the same time. This is made explicit in the redefined grammar. Furthermore, the order of appearance for pre- and postconditions has been defined. Moreover, empty context declarations are allowed when neither the keyword `self`, nor any alternative single object occurs within the constraint expression.

```
⟨constraint⟩ ::=  context ⟨clContext⟩ inv { ⟨name⟩ }? : ⟨exprList⟩
                | context ⟨opContext⟩
                  { pre { ⟨name⟩ }? : ⟨exprList⟩ }?
                  { post { ⟨name⟩ }? : ⟨exprList⟩ }?
                | ⟨exprList⟩
 ⟨clContext⟩ ::= { ⟨name⟩ : }? ⟨class⟩
 ⟨opContext⟩ ::= { ⟨name⟩ : }? ⟨class⟩ :: ⟨methodName⟩ ( ⟨fParamList⟩? )
                 { : ⟨typeExpr⟩ }?
⟨fParamList⟩ ::= ⟨fParam⟩ { , ⟨fParam⟩ }*
    ⟨fParam⟩ ::= ⟨varName⟩ : ⟨typeExpr⟩
```

The OCL specification [16] allows an alternative name for `self` within the class context specification, whereas within the operation context specification such an alternative is not allowed. This can be more systematic, by allowing alternative names in both cases.

To reduce the number of required non-terminals and grammar rules, the precedence rules for operations are given by a separate table. A complete precedence rules table is given separately in Section 5.3. As already discussed in Section 2 these could be expressed within grammar rules as well, but their presentation is more compact and readable within the precedence rules table.

```
     ⟨expr⟩ ::= ⟨unaryExpr⟩ { ⟨infixOp⟩ ⟨unaryExpr⟩ }*
⟨unaryExpr⟩ ::= ⟨unaryOp⟩* ⟨primeExpr⟩
  ⟨unaryOp⟩ ::=  - | not
  ⟨infixOp⟩ ::= and | or | xor | implies
              | = | > | < | >= | <= | <> | + | - | * | /
```

Unfortunately, the description of ⟨primeExpr⟩ and related non-terminals in [16] is rather confusing. This makes it quite difficult for the reader to understand

what exactly expressions are and what their meaning is. After a detailed analysis of ⟨primeExpr⟩ the following restructuring was developed. Due to its many variants, the grammar rules for ⟨primeExpr⟩ are presented in several parts:

```
⟨primeExpr⟩ ::= ⟨letExpr⟩
              | ⟨ifExpr⟩
              | ( ⟨expr⟩ )
              | ⟨collectKind⟩ { { ⟨collItem⟩ { , ⟨collItem⟩ }* }? }
              | ⟨property⟩
              | ⟨primeExpr⟩ . ⟨property⟩
              | ⟨primeExpr⟩ . ⟨collectFeature⟩
              | ...
```

In [16], the literal collection allows either a single range of values or a list of individual values, but not all in one, e.g. Set {1, 4..7, 11..13, 2}. The grammar was extended accordingly, by enhancing the ⟨collItem⟩ later on.

The original specification uses an arrow symbol `->` for calling a collection operation. The operations of the other types however, are called like a usual property with a dot between both arguments, like e.g. `i.abs` (absolute value of an integer `i`). This lack of uniformity adds unnecessary complexity to OCL. This complexity was removed by replacing the arrow with a dot. The often heard argument, that it is necessary to distinguish normal features from OCL-features using arrows, simply does not hold, because a type system can easily resolve name clashes.

```
⟨primeExpr⟩ ::= -- continued
              | ⟨type⟩
              | ⟨literal⟩
              | self
              | result
              | ...
```

Not only constants, but also the special keywords `self` and `result`, types, and in particular class names may be used at expression position. A class name used as expression is interpreted as the set of all existing objects of that class at that time. This replaces the superfluous OCL feature `allInstances`. Class names can then be used in two functions. They are allowed, both where types or set values are appropriate and their position can be resolved because of a well structured grammar. As the new grammar distinguishes between ⟨typeExpr⟩ and ⟨expr⟩, this approach works.

⟨primeExpr⟩ ::= -- continued
| ⟨primeExpr⟩ **.oclAsType** (⟨typeExpr⟩)
| ⟨primeExpr⟩ **.oclIsKindOf** (⟨typeExpr⟩)
| ⟨typeifExpr⟩
| ⟨primeExpr⟩ **.oclIsInState** (⟨stateName⟩)
| ⟨primeExpr⟩ **.oclIsNew**
| ...

The special operation `p.oclAsType(T)` means the type of expression `p` is changed to `T`. The second special operation `p.oclIsKindOf(T)` evaluates to true exactly if the value denoted by `p` is of type or a subtype of `T`.

The four OCL specific constructs `oclIsKindOf`, `oclAsType`, `oclIsInState`, and `oclIsNew` are explicitly included in the ⟨primeExpr⟩ grammar rule. This shows that these properties are not normal functions, but special *control constructs*. For example, in programming languages like Java, the casting of objects differs syntactically from ordinary method invocation. Unfortunately, OCL syntax did not distinguish this and, therefore, OCL is confusing here, because it tried to fit every concept into method call structure. Now this principal distinction is demonstrated at least in the presentation of the syntax. Furthermore, this rearrangement makes it unnecessary to treat their arguments as objects. Therefore, meta-level type `OclType` is not needed here anymore.

Unfortunately the casting construct `oclAsType` has the drawback that it needs to deal with cast failures. This may not result in a raised exception, as logic doesn't raise exceptions at all. The newly introduced alternative construct ⟨typeifExpr⟩ combines the cast with an `if-then-else` construction that allows to specify both cases and should largely replace `oclAsType` (see below).

In the last part, timed expressions, qualified associations and usage of roles are defined in the same way as it had been given in original OCL.

⟨primeExpr⟩ ::= -- continued
| ⟨primeExpr⟩ **@pre**
| ⟨primeExpr⟩ [⟨exprList⟩]
| ⟨primeExpr⟩ [⟨roleName⟩]

The definition of the `let` expression is extended by multiple bindings. It is now possible to declare more than one variable at the same time. Furthermore, a `let` expression can be used within other `let` expressions.

The ⟨typeifExpr⟩ rule allows a variable to be introduced and bound to an expression. The variable has the casted type in the then-part and its original type in the else-part.

⟨letExpr⟩ ::= **let** ⟨declList⟩ **in** ⟨expr⟩

⟨ifExpr⟩ ::= **if** ⟨expr⟩ **then** ⟨expr⟩ **else** ⟨expr⟩ **endif**

⟨typeifExpr⟩ ::= **typeif** ⟨decl⟩ **isof** ⟨typeExpr⟩ **then** ⟨expr⟩ **else** ⟨expr⟩ **endif**

⟨property⟩ ::= ⟨attributeName⟩
| ⟨methodName⟩ (⟨exprList⟩?)
| ⟨roleName⟩

⟨literal⟩ ::= ⟨char⟩ | ⟨string⟩ | ⟨number⟩ | ⟨bool⟩
| ⟨enumTypeName⟩ :: ⟨name⟩

Predefined operations applicable to the basic types `Integer`, `Real`, `String` and `Character`, such as `abs`, `floor`, or `size` are treated like ordinary method calls. In case they don't have arguments, they are like attributes. See [16] for a full list of operators.

In [16], the possibility of specifying with Boolean values was missing and has been added here.

The definition of the enumeration types in [16] exhibits some inconsistency when compared to the UML enumerations. All enumeration types and their values must be defined in the underlying UML model. Therefore, each enumeration type already has a name. The redefined grammar uses the same notation as for class pathnames to refer to a certain enumeration type. If the enumeration type `Color` among others contains the element `red`, then `Color::red` is a qualified enumeration value.

The nonterminal ⟨collectFeature⟩ describes the features of collection types. For brevity in the following only a few examples of each category are listed, i.e. operations without parameter, operations with one parameter of type ⟨expr⟩, but all the special constructs with their specific parameters. A complete list of the normal features can be inferred from the official OCL definitions.

⟨collectFeature⟩ ::= **size** | **isEmpty** | ...
| **includes** (⟨expr⟩) | **union** (⟨expr⟩) | ...
| **select** (⟨var⟩ | ⟨expr⟩)
| **reject** (⟨var⟩ | ⟨expr⟩)
| **collect** (⟨var⟩ | ⟨expr⟩)
| **forAll** (⟨varList⟩ | ⟨expr⟩)
| **exists** (⟨varList⟩ | ⟨expr⟩)
| **iterate** (⟨var⟩ ; ⟨decl⟩ | ⟨expr⟩) | ...

The special constructs `select`, etc. have been redefined in their structure. They are not treated as ordinary functions anymore, but explicitly introduced

through the grammar. In the presented grammar each of them has a *block* as body. A block at first introduces and binds one or several new variables and then allows to express a constraint over that variables. So far the OCL definition has treated these bodies wrongly as ordinary arguments. This adaptation is a good example for a grammar restructuring that does not affect the language, but makes the language presentation more conform to standard language definitions. But finally these constructs have got two changes: In official OCL it was allowed to omit the explicit introduction of a variable, in which case the variable `self` was newly introduced and bound implicitly. This can easily lead to misunderstanding in more complex formulae and is therefore abandoned. On the other hand, for some of these operators it is now possible to introduce several variables at once.

The non-terminals used in the above control structures look like this:

⟨varList⟩ ::= ⟨var⟩ { , ⟨var⟩ }*
⟨var⟩ ::= ⟨varName⟩ { : ⟨typeExpr⟩ }?
⟨declList⟩ ::= ⟨decl⟩ { , ⟨decl⟩ }*
⟨decl⟩ ::= ⟨var⟩ = ⟨expr⟩
⟨exprList⟩ ::= ⟨expr⟩ { , ⟨expr⟩ }*
⟨collItem⟩ ::= ⟨expr⟩ { .. ⟨expr⟩ }?

Unlike the grammar of [16] this grammar explicitly allows class names to be qualified by a package name in the non-terminal ⟨class⟩. This guarantees that a package name may be given wherever ⟨class⟩ appears. This feature is essential for dealing with identical class names in different packages.

⟨class⟩ ::= {⟨packageName⟩ :: }* ⟨className⟩
⟨collectKind⟩ ::= **Set** | **Bag** | **Sequence** | **Collection**
⟨typeExpr⟩ ::= ⟨collectKind⟩ (⟨type⟩) | ⟨type⟩
⟨type⟩ ::= ⟨class⟩ | ⟨basicType⟩ | ⟨enumTypeName⟩
⟨basicType⟩ ::= **Integer** | **Real** | **Boolean** | **String** | **Character**

The well known and often used type `Character` was missing in OCL and therefore newly added to the basic types. Its operations are given in 5.2.

The rule for numbers that allows floating-point constants is extended, because numeric constants were missing in [16]:

```
 ⟨digit⟩ ::= [ 0 − 9 ]
⟨digits⟩ ::= ⟨digit⟩ { ⟨digit⟩ }*
⟨number⟩ ::= ⟨digits⟩ { . ⟨digits⟩ }? { { e | E } { + | − }? ⟨digits⟩ }?
⟨letter⟩ ::= [ a − z A − Z _ ]
  ⟨name⟩ ::= ⟨letter⟩ { ⟨letter⟩ | ⟨digit⟩ }*
  ⟨char⟩ ::= ' valid Unicode character '
⟨string⟩ ::= " ⟨char⟩* "
  ⟨bool⟩ ::= True | False
```

The resulting language is now rather conform to standard ways of presenting expression syntax. It indeed improves readability and eases tool implementation.

5 Additional adaptations based on the new grammar

The previously defined context free grammar describes a language quite similar to the official OCL. It enhances or restricts some constructs, but largely focuses on better presentation of its context free grammar.

As a next step, it is necessary to adapt the context conditions to the new grammar where necessary. This will not be done in this article, but a small set of these issues will be tackled here.

5.1 Type hierarchy

In the OCL specification [16] `OclAny` is a supertype of both model and basic types. On the one hand, this gives rise for some subtle type conformance problems. On the other hand, it would be more convenient to have the possibility to address the universe of all objects, without basic values such as numbers included. Therefore, `OclAny` now is the supertype of all class types, but excludes the basic types and all collection types.

This also means, that `Set(OclAny)` is not included in `OclAny` anymore, but the universe of sets, distinct from the universe of objects. The corrected type hierarchy is given in Fig. 3.

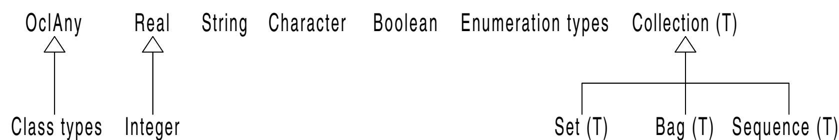

Figure 3. OCL type hierarchy

5.2 Predefined operations

As already mentioned, logic level and meta-level are strictly separated. In the grammar as given in [16], the basic types `Integer`, `Real`, `String` and `Boolean` are on the same level as OCL specific meta-types, namely `OclExpression` and `OclType`. These meta-types were used in the grammar to describe specific kinds of arguments for a number of special constructs. After the restructuring presented in this paper, these types become superfluous. Access to the meta-level is no longer necessary. Therefore, `OclExpression` and `OclType` can be removed.

Please note that parentheses are used within the definitions of the infix operations to improve readability.

OclAny Here, `object` is a variable of type `OclAny`. `OclAny` has the following signature:

```
object = (object1 : OclAny) : Boolean
object <> (object1 : OclAny) : Boolean
```

Both operations are heterogeneous. Therefore, variables `object` and `object1` do not need to be of the same type, but may be arbitrary class types.

Basic types For the sake of brevity only a few `Integer` operators are shown.

Due to the type system re-arrangements, the overall equality does only accept class types as arguments, but not basic types. Therefore, several additional equality functions on each basic type are needed. These functions can be additionally defined, because in typed languages the syntactic overloading of methods can be resolved statically. This means the compiler can determine, which equality is to be used.

Here, `i` is a variable of type `Integer`.

```
i = (i1 : Integer) : Boolean
i <> (i1 : Integer) : Boolean
i + (i1 : Integer) : Integer
...
```

Enumeration types Enumerations defined in a UML model have to be represented within OCL constraints. There may exist various different enumerations in a model. Therefore, the attempt of [16] to describe all enumerations with only one enumeration type does not work properly. Instead, enumerations are treated as individual, non-further related types. In UML, each enumeration is defined as a special datatype and accordingly has a type name and the belonging values. Within OCL an enumeration value is referred to by the enumeration type name and the value name. Let `enum` be a variable of a given enumeration type `E`.

```
E::enum = (E::enum1 : E) : Boolean
E::enum <> (E::enum1 : E) : Boolean
```

In languages like C++, further functionality exists for enumeration types: for example, a linear order or a successor function.

Please note that the built-in type Boolean can be regarded as a two-valued enumeration type.

Collection types Below the signatures of collection types are introduced as polymorphic types. The key idea is that if an operation *doesn't care* about the value type, then it reacts uniformly regardless of what these values are.

This section deals with the predefined features on the parameterized collection types `Collection(T)`, `Set(T)`, `Bag(T)`, and `Sequence(T)`, where the type variable `T` denotes any type except collection types.

In the grammar provided by this article, these features are included by the construct

⟨primeExpr⟩ . ⟨collectFeature⟩

As already mentioned, in the grammar of [16] all possible features for collections were treated as method calls. This was not quite correct. In fact, they mix both method calls and specialized control structures. The grammar defined above realizes this difference, because the control structures are explicitly mentioned in the grammar. This now allows to explicitly mention the variable lists in the grammar that are separated from the normal expression syntax.

Collection-based control structures usually evaluate a particular expression for each element in the collection. Again inspired by Smalltalk the syntax of these control structures as defined in [16] is somewhat misleading. According to [16] the argument of such a control structure is of type `OclExpression`. As the redefinition of the grammar already shows that the binding of new variables should be allowed in blocks consisting of a declarative part and an expression part.

A block is an introduction of a new variable together with an expression that uses the variable. The standard syntax for the `forAll` function:

```
Collection(T).forAll(t : T | expr : Boolean)
```

Quantification can range over a number of variables:

```
Collection(T).forAll(t1 : T, ... , tn : T | expr : Boolean)
```

A block consists of one or several defined variables and then, separated by a vertical bar |, a Boolean expression follows. Functional languages would regard this block as a function definition based on a λ-abstraction.

OCL from [16] does allow to omit an explicit introduction of a variable. In this case, the block looks like an expression, but it still serves the duties of a block, as implicitly the variable `self` is introduced. As already mentioned, the restructured grammar does not allow implicit binding of `self` anymore.

5.3 Precedence Rules

For the reasons of completeness, a list of precedence rules is included here. Although the rules can be modeled by the grammar itself, it is more compact and equally precise to use an explicit precedence table.

Name	Syntax	Associativity
Pathname	`::`	left
Time expression	`@pre`	left
Dot operation	`.`	left
Unary operations	`-, not`	right
Multiplication and division	`*, /`	left
Addition and subtraction	`+, -`	left
Relational operations	`<, >, <=, >=`	left
Relational equality	`=, <>`	left
Logical operations	`and, or, xor`	left
Logical implies	`implies`	right

Table 2. Precedence for OCL operations (highest to lowest).

The associativity of an operator specifies the order that operations of the same precedence are performed in. Left associativity means that operations are grouped from left-to-right. For example:

```
a.b.c                   = (a.b).c
a / b / c               = (a / b) / c
a and b and c           = (a and b) and c
a implies b implies c = a implies (b implies c)
```

Please note that dependent on the semantics of the `and` operator, it is not necessarily the case that `and` is associative, i.e. according to Fig. 2 it can be wrong to replace

`a and (b and c)` by `(a and b) and c`

5.4 Summary of grammar adaptations

Three categories of adaptations can be identified. Some grammar adaptations affect the language, other changes affect the predefined operations, and a third group is introduced through adaptations of the context conditions. This is a short summary of the according changes. The new grammar was compared to [16].

Grammar based language changes Often, but not always, changes of the grammar effect the language. A number of such grammar based language changes is listed below.

- The consistency of the grammar was improved: previously some nonterminals were enclosed in angle brackets, and others not.

- Separation of invariants and pre-/postconditions within the constraint definition.
- Clarification of the order in which pre- and postconditions appear.
- Correction of the class context specification. It is now possible to deal with identical class names in different packages by qualifying the class context specification with an optional pathname.
- The grammar of the context declaration was enhanced, allowing an alternative name for `self` also for the operation context.
- The type of method parameters as well as the return type of a method was changed to ⟨typeExpr⟩, which additionally contains collection and enumeration types. This allows the underlying UML model to provide methods for OCL constraint specifications with that according types.
- Method parameters are now separated by comma instead of semicolon.
- Precedence rules have been removed from the grammar and given in a precedence table.
- Unary operators are now allowed to occur repeatedly.
- The grammar for the non-terminal ⟨primeExpr⟩ was largely restructured.
- The syntax for the literal collection was changed to allow more general enumeration and number constants.
- The arrow symbol, used for calling a collection operation, was replaced by a dot.
- The let-expression was extended so that firstly, multiple bindings are allowed and secondly, more than one let-expression may occur in an expression.
- Nonterminal ⟨literal⟩ now contains the Boolean values.
- The definition of the enumeration types was corrected, since in [16] it is inconsistent with the UML meta-model.
- Whenever a class name is used, this class can be qualified using a path name.
- Numbers now include floating-point constants.
- Along with the introduction of the new type `Character` the notation of strings was adapted. In the redefined grammar, strings have to be enclosed by double quotes.

Change of the predefined operations

- The operator <> was added to the operations of Integer, String, Boolean and the three concrete collection types.
- `Integer` and `String` were additionally completed by the operations <, <=, >, and >=.
- The operations on collection types where divided in method calls and control structures.
- The block concept was introduced for the control structures.
- For the operations `includes`, `excludes` and `count` the type of the called argument was changed to the type of the collection elements.
- The `forAll` operation has an extended variant.

Change of the context conditions

- The type hierarchy of OCL was adapted: the type `OclAny` is not a supertype of the basic types or the collection types anymore.
- For a strict separation of logic level and meta-level, the meta types `OclType` and `OclExpression` have been removed.
- The explicit possibility of building all type instances was removed. Instead of `allInstances` the class name can be used directly now.
- The type `Character` was newly introduced.

6 ≪Java≫OCL: The Java-variant of OCL

≪Java≫OCL has almost the same abstract syntax as standard OCL. However, variable declarations, method calls, type casts and some other concepts have been given a different syntactic shape. ≪Java≫OCL is conform to Java and therefore is more familiar to Java developers. Some small differences occur due to available operations and e.g. due to the change of the cast into an infix operation.

For a detailed description of the differences, the previously defined OCL grammar and the new ≪Java≫OCL grammar are listed on the left resp. right hand side. Small adaptations are explained below.

OCL	≪Java≫OCL
⟨constraint⟩ ::=	
context ⟨clContext⟩ **inv** { ⟨name⟩}$^{?}$: ⟨exprList⟩ | **context** ⟨opContext⟩ { **pre** { ⟨name⟩ }$^{?}$: ⟨exprList⟩ }$^{?}$ { **post** { ⟨name⟩ }$^{?}$: ⟨exprList⟩ }$^{?}$ | ⟨exprList⟩	**context** ⟨clContext⟩ **inv** { ⟨name⟩}$^{?}$: ⟨exprList⟩ | **context** ⟨opContext⟩ { **pre** { ⟨name⟩ }$^{?}$: ⟨exprList⟩ }$^{?}$ { **post** { ⟨name⟩ }$^{?}$: ⟨exprList⟩ }$^{?}$ | ⟨exprList⟩
⟨clContext⟩ ::=	
{ ⟨name⟩ : }$^{?}$ ⟨class⟩	⟨class⟩ { ⟨name⟩ }$^{?}$
⟨opContext⟩ ::=	
⟨clContext⟩ :: ⟨methodName⟩ (⟨fParamList⟩$^{?}$) { : ⟨typeExpr⟩ }$^{?}$	{ ⟨typeExpr⟩ }$^{?}$ ⟨class⟩ . ⟨methodName⟩ (⟨fParamList⟩$^{?}$)
⟨fParamList⟩ ::=	
⟨fParam⟩ { , ⟨fParam⟩ }*	⟨fParam⟩ { , ⟨fParam⟩ }*
⟨fParam⟩ ::=	
⟨varName⟩ : ⟨typeExpr⟩	⟨typeExpr⟩ ⟨varName⟩

Due to the switched nonterminals in the ⟨clContext⟩ rule, it becomes confusing to allow relabelling of the object in the ⟨opContext⟩. Therefore, this is omitted in ≪Java≫OCL.

<table>
<tr><th>OCL</th><th>≪Java≫OCL</th></tr>
<tr><td colspan="2">⟨expr⟩ ::=</td></tr>
<tr><td>⟨unaryExpr⟩ { ⟨infixOp⟩ ⟨unaryExpr⟩ }*</td><td>⟨unaryExpr⟩ { ⟨infixOp⟩ ⟨unaryExpr⟩ }*</td></tr>
<tr><td colspan="2">⟨unaryExpr⟩ ::=</td></tr>
<tr><td>⟨unaryOp⟩* ⟨primeExpr⟩</td><td>⟨unaryOp⟩* ⟨primeExpr⟩</td></tr>
<tr><td colspan="2">⟨unaryOp⟩ ::=</td></tr>
<tr><td>- | not</td><td>- | not
| ~ | !
| (⟨typeExpr⟩)</td></tr>
<tr><td colspan="2">⟨infixOp⟩ ::=</td></tr>
<tr><td>and | or | xor | implies
| = | > | < | >= | <= | <>
| + | - | * | /</td><td>&& | || | xor | implies
| == | > | < | >= | <= | !=
| + | - | * | /</td></tr>
</table>

An additional unary operator has been added that allows type casts. It replaces the `oclAsType` operation. New binary operators stem from additional functionality on basic Java types.

<table>
<tr><th>OCL</th><th>≪Java≫OCL</th></tr>
<tr><td colspan="2">⟨primeExpr⟩ ::=</td></tr>
<tr><td>⟨letExpr⟩
| ⟨ifExpr⟩
| (⟨expr⟩)
| ⟨collectKind⟩
{ { ⟨collItem⟩ {, ⟨collItem⟩ }* }? }
| ⟨property⟩
| ⟨primeExpr⟩ . ⟨property⟩

| ⟨primeExpr⟩ . ⟨collectFeature⟩

| ⟨type⟩
| ⟨literal⟩
| self
| result
| ⟨primeExpr⟩ .oclAsType
(⟨typeExpr⟩)
| ⟨primeExpr⟩ .oclIsKindOf
(⟨typeExpr⟩)
| ⟨typeifExpr⟩
| ⟨primeExpr⟩ .oclIsInState
(⟨stateName⟩)
| ⟨primeExpr⟩ .oclIsNew
| ⟨primeExpr⟩ @pre
| ⟨primeExpr⟩ [⟨exprList⟩]
| ⟨primeExpr⟩ [⟨roleName⟩]</td><td>⟨letExpr⟩
| ⟨ifExpr⟩
| (⟨expr⟩)
| ⟨collectKind⟩
{ { ⟨collItem⟩ {, ⟨collItem⟩ }* }? }
| ⟨property⟩
| ⟨primeExpr⟩ . ⟨property⟩
| ⟨class⟩ . ⟨methodName⟩ (⟨exprList⟩?)
| ⟨primeExpr⟩ . ⟨collectFeature⟩
| ⟨collectExpr⟩
| ⟨type⟩
| ⟨literal⟩
| this
| result

| ⟨primeExpr⟩ instanceOf ⟨typeExpr⟩

| ⟨typeifExpr⟩
| ⟨primeExpr⟩ .isInState (⟨stateName⟩)

| ⟨primeExpr⟩ .isNew
| ⟨primeExpr⟩ '
| ⟨primeExpr⟩ [⟨exprList⟩]
| ⟨primeExpr⟩ [⟨roleName⟩]</td></tr>
</table>

As there are many OCL specific operations that don't have "ocl" as prefix, this was also removed from the above mentioned special operations. A useful alternative might be to write `OCL.isNew(o)` instead of `o.isNew`. As Java generally allows to use static methods, this possibility was added to ≪Java≫OCL as

well. Then a number of OCL specific operations might be provided by a special "class" called OCL.

The new nonterminal ⟨collectExpr⟩ was added, to allow a syntactic differentiation between method calls and special constructs such as `forall`.

<table>
<tr><th>OCL</th><th>≪Java≫OCL</th></tr>
<tr><td colspan="2">⟨letExpr⟩ ::=</td></tr>
<tr><td>let ⟨declList⟩ in ⟨expr⟩</td><td>let ⟨declList⟩ in ⟨expr⟩</td></tr>
<tr><td colspan="2">⟨ifExpr⟩ ::=</td></tr>
<tr><td>if ⟨expr⟩ then ⟨expr⟩ else ⟨expr⟩ endif</td><td>if ⟨expr⟩ then ⟨expr⟩ else ⟨expr⟩
| ⟨expr⟩ ? ⟨expr⟩ : ⟨expr⟩</td></tr>
<tr><td colspan="2">⟨typeifExpr⟩ ::=</td></tr>
<tr><td>typeif ⟨decl⟩ isof ⟨typeExpr⟩
then ⟨expr⟩ else ⟨expr⟩ endif</td><td>typeif ⟨decl⟩ instanceof ⟨typeExpr⟩
then ⟨expr⟩ else ⟨expr⟩
| ⟨decl⟩ instanceof ⟨typeExpr⟩
? ⟨expr⟩ : ⟨expr⟩</td></tr>
<tr><td colspan="2">⟨property⟩ ::=</td></tr>
<tr><td>| ⟨attributeName⟩
| ⟨methodName⟩ (⟨exprList⟩?)
| ⟨roleName⟩</td><td>| ⟨attributeName⟩
| ⟨methodName⟩ (⟨exprList⟩?)
| ⟨roleName⟩</td></tr>
<tr><td colspan="2">⟨literal⟩ ::=</td></tr>
<tr><td>⟨char⟩ | ⟨string⟩ | ⟨number⟩ | ⟨bool⟩

| ⟨enumTypeName⟩ :: ⟨name⟩</td><td>⟨boolean⟩ | ⟨char⟩ | ⟨int⟩
| ⟨String⟩ | ...
| ⟨class⟩ . ⟨name⟩</td></tr>
</table>

In Java an `endif` is not used. Therefore, in nested `if`-expressions it might become necessary to add brackets. The `.?.:.` variant for expressions is added as additional alternative for `if`-expressions. Java handles enumeration types as ordinary integer constants. However, a tricky way of denoting these constants gives a look and feel as if they would be true enumeration types. ≪Java≫OCL handles enumeration types in a similar manner.

<table>
<tr><th>OCL</th><th>≪Java≫OCL</th></tr>
<tr><td colspan="2">⟨collectFeature⟩ ::=</td></tr>
<tr><td>size| isEmpty | ...
| includes (⟨expr⟩)
| ... | union (⟨expr⟩)
| select (⟨var⟩ | ⟨expr⟩)
| reject (⟨var⟩ | ⟨expr⟩)
| collect (⟨var⟩ | ⟨expr⟩)
| forAll (⟨varList⟩ | ⟨expr⟩)
| exists (⟨varList⟩ | ⟨expr⟩)
| iterate (⟨var⟩ ; ⟨decl⟩ | ⟨expr⟩) | ...</td><td>size| isEmpty | ...
| includes (⟨expr⟩)
| ... | union (⟨expr⟩)</td></tr>
</table>

Special constructs, like `forall` are rearranged such that their syntax immediately differs from method calls. Previously `select` was written as

```
Person.select( p | p.age > 5 )
```

and is now formulated as

```
select p=Person | p.age > 5
```

as well as `(select p=Person | p.age > 5)`. Inversion of the first argument makes the constructs much more flexible, as it is now possible to use different expressions for different variables within one quantifier:

```
exists p=Person, f=p.father | f.age < p.age + 18
```

<table>
<tr><th>OCL</th><th>≪Java≫OCL</th></tr>
<tr><td></td><td>⟨collectExpr⟩ ::=
  select ⟨decl⟩ | ⟨expr⟩
| reject ⟨decl⟩ | ⟨expr⟩
| collect ⟨decl⟩ | ⟨expr⟩
| forAll ⟨declList⟩ | ⟨expr⟩
| exists ⟨declList⟩ | ⟨expr⟩
| iterate (⟨var⟩ , ⟨decl⟩) ⟨expr⟩</td></tr>
</table>

The `iterate` operator resembles a Java `while/for` loop as close as possible.

<table>
<tr><th>OCL</th><th>≪Java≫OCL</th></tr>
<tr><td>⟨varList⟩ ::=
⟨var⟩ { , ⟨var⟩ }*</td><td>
⟨var⟩ { , ⟨var⟩ }*</td></tr>
<tr><td>⟨var⟩ ::=
⟨varName⟩ { : ⟨typeExpr⟩ }?</td><td>
{ ⟨typeExpr⟩ }? ⟨varName⟩</td></tr>
<tr><td>⟨declList⟩ ::=
⟨decl⟩ { , ⟨decl⟩ }*</td><td>
⟨decl⟩ { , ⟨decl⟩ }*</td></tr>
<tr><td>⟨decl⟩ ::=
⟨var⟩ = ⟨expr⟩</td><td>
⟨var⟩ = ⟨expr⟩</td></tr>
<tr><td>⟨exprList⟩ ::=
⟨expr⟩ { , ⟨expr⟩ }*</td><td>
⟨expr⟩ { , ⟨expr⟩ }*</td></tr>
<tr><td>⟨collItem⟩ ::=
⟨expr⟩ { .. ⟨expr⟩ }?</td><td>
⟨expr⟩ { .. ⟨expr⟩ }?</td></tr>
</table>

<table>
<tr><th>OCL</th><th>≪Java≫OCL</th></tr>
<tr><td>⟨class⟩ ::=
{⟨packageName⟩ :: }* ⟨className⟩</td><td>
{⟨packageName⟩ . }* ⟨className⟩</td></tr>
<tr><td>⟨collectKind⟩ ::=
Set | Bag | Sequence | Collection</td><td>
Set | Bag | Sequence | Collection</td></tr>
<tr><td>⟨typeExpr⟩ ::=
⟨collectKind⟩ (⟨type⟩) | ⟨type⟩</td><td>
⟨collectKind⟩ (⟨type⟩) | ⟨type⟩</td></tr>
<tr><td>⟨type⟩ ::=
⟨class⟩ | ⟨basicType⟩ | ⟨enumTypeName⟩</td><td>
⟨class⟩ | ⟨basicType⟩</td></tr>
<tr><td>⟨basicType⟩ ::=
Integer | ...</td><td>
int | ...</td></tr>
</table>

Of course Boolean values, digits etc. differ between OCL and Java, but their detailed definition is omitted here. Further changes appear in the context conditions. For example the type `OclAny` should be replaced by Java's `Object`.

7 Conclusion

In this article, a number of syntactical and semantic inconsistencies of OCL have been revealed and corrected. One important change concerns the strict

separation of logic level and meta-level. Furthermore, the given grammar of the OCL specification [16] was changed in a manner that achieves an improved grammar structure, allowing easier readability and a better implementation of tools. The improved OCL presentation using the restructured grammar is now more in line with standard programming language grammars. Accordingly, it allows an easier comparison and translation of OCL to standard programming languages. Furthermore, this conformance increases readability.

In a last step «Java»OCL was introduced as a Java-like variant of OCL. It has (almost) the same abstract syntax as the enhanced version of OCL presented in this paper, but its syntactic sugar is given in the flavor of Java.

Acknowledgements

This work was partially funded by the Bayerisches Staatsministerium für Wissenschaft, Forschung und Kunst under the Habilitation-Förderpreis program and by the Bayerische Forschungsstiftung under the FORSOFT research consortium. Many thanks go to my colleagues Manfred Broy, Peter Braun, Steve Cook, Heinrich Hußmann, Jos Warmer, and in particula Manuela Scherer for fruitful input and discussion on this topics.

References

1. Aliprand, J., Allen, J., Becker, J., Davis, M., Everson, M., Freytag, A., Jenkins, J., Ksar, M., McGowan, R., Moore, L., Suignard, M., Whistler, K., *The Unicode Standard, Version 3.0.* Addison Wesley Longman Publisher, 2000.
2. Boger, M., Baier, T., Wienberg, F., *Extreme Modeling*, In: XP'2000 conference proceedings, Ed. Michele Marchesi (to appear), Addison-Wesley, 2001.
3. Broy, M., Facchi, C., Grosu, R., Hettler, R., Hussmann, H., Nazareth, D., Regensburger, F., Slotosch, O., Stolen K., *The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part ii* . Technical Report TUM-I9312, Technische Universität München. Institut für Informatik, TUM, München, Germany, May 1993.
4. Church A, *A formulation of the simple theory of types.* Journal of Symbolic Logic, 5:56-68, 1940.
5. Cook, S., Kleppe, A., Mitchell, R., Rumpe, B., Warmer, J., Wills, A., *The Amsterdam Manifesto on OCL*, Technical Report, Technische Universität München, Computer Science, 1999.
6. Cook, S., Daniels, J. *Designing Object Systems–Object Oriented Modeling with Syntropy.* Prentice-Hall, 1994.
7. Fitzgerald, J., Larsen, P. G., *Modelling Systems: Practical Tools and Techniques in Software Development.* Cambridge University Press, 1998.
8. Gogolla, M., Richters, M., *On Constraints and Queries in UML.* In M. Schader and A. Korthaus, editors, *Proc. UML'97 Workshop 'The Unified Modeling Language - Technical Aspects and Applications'*, pages 109-121. Physica-Verlag, Heidelberg, 1997.
9. Gosling, J,. Joy, B., Steele, G., *The Java Language Specification*, Addison-Wesley, 1996.

10. Hamie, A., Civello, F., Howse, J., Kent, S., Mitchell, R., *Reflections on the Object Constraint Language.* In P. Muller and J. Bézivin, editors, Proc. of UML'98 International Workshop, Mulhouse, France, June 3-4, 1998, pages 137-145.
11. Hamie, A., Howse, J., Kent S., *Interpreting the Object Constraint Language.* In Proceedings of Asia Pacific Conference in Software Engineering. IEEE Press, July 1998.
12. Harel, D., Rumpe, B., *Modeling Languages: Syntax, Semantics and All That Stuff*, The Weizmann Institute of Science, Rehovot, Israel, MCS00-16, 2000.
13. Hussmann, H., OCL Compiler. Technische Universität Dresden. Available from `http:// www-st.inf.tu-dresden.de/ocl`, 2001.
14. Jacobi, C., Rumpe, B., *Hierarchical XP – Improving XP for large scale projects*, In: XP'2000 conference proceedings, Ed. Michele Marchesi (to appear), Addison-Wesley, 2001.
15. OCL Parser, Version 0.3. Available from `http://www.software.ibm.com/ad/standards/ocl-download.html`, 1999.
16. OMG. Object Constraint Language Specification. In *OMG Unified Modeling Language Specification, Version 1.3 (June 1999)*, chapter 7. Available from `http://www.rational.com`.
17. OMG Unified Modeling Language Specification, Version 1.3 (June 1999). Available from `http://www.rational.com`.
18. Richters, M., Gogolla, M., *On Formalizing the UML Object Constraint Language OCL.* In Tok-Wang Ling, Sudha Ram, and Mong Li Lee, editors, Proc. 17th Int. Conf. Conceptual Modeling (ER'98), pages 449-464. Springer, Berlin, LNCS 1507, 1998.
19. Rumpe, B., *Gofer Objekt-System – Imperativ Objektorientierte und Funktionale Programmierung in einer Sprache*, Technical Report, Universität Passau, MIP-9519, 1995.
20. Warmer, J., Kleppe, A., *The Object Constraint Language: Precise Modeling with UML.* Addison Wesley Longman, Reading, Massachusetts, 1999.
21. Wirth, N., *Programming in Modula-2*, Springer Verlag, 1982.