Abstract
This document describes the specification and implementation of an UML (Unified Modeling Language) tool supporting Agile MDA (Model Driven Architecture).
AGILE MDA

Presentation
Validation
Debugging
Generation
Table of Contents

1 Introduction .................................................................................................................. 1
  1.1 UML Standardization through OMG ................................................................. 1
  1.2 Positioning the Object Constraint Language (OCL) ........................................... 1

2 The Architecture of the Agile MDA Tool .................................................................... 3

3 Transformations and Code Generations .................................................................... 4
  3.1 Model Derivations with Transformations ............................................................ 4
  3.2 Code Generators .................................................................................................. 5
  3.3 OCL Extensions .................................................................................................... 5
    3.3.1 Context Declarations for Transformations .................................................. 5
    3.3.2 Context Declarations for Code Generations ................................................. 7
    3.3.3 Variable Type "TextFile" ............................................................................... 8
    3.3.4 Predefined (Library) Operations .................................................................. 8

4 The OCL Library ......................................................................................................... 9
  4.1 Operations on UML model elements .................................................................... 9
  4.2 Operations on Primitive Types .......................................................................... 9
    4.2.1 Real Type Operations .................................................................................. 10
    4.2.2 Integer Type Operations .......................................................................... 10
    4.2.3 String Type Operations .............................................................................. 11
    4.2.4 Boolean Type Operations .......................................................................... 15
    4.2.5 TextFile Type Operations .......................................................................... 15
  4.3 Operations on Collection Types .......................................................................... 16
    4.3.1 Collection Operations ................................................................................. 17
    4.3.2 OrderedSet Operations ............................................................................... 17
    4.3.3 Set Operations ............................................................................................. 18
    4.3.4 Sequence Operations .................................................................................. 19
    4.3.5 Bag Operations ............................................................................................ 20
  4.4 Iterators .................................................................................................................. 21
    4.4.1 Collection Iterators ..................................................................................... 21
    4.4.2 OrderedSet Iterators ................................................................................... 22
    4.4.3 Set Iterators .................................................................................................. 22
    4.4.4 Sequence Iterators ...................................................................................... 23
    4.4.5 Bag Iterators ................................................................................................. 23
    4.4.6 The Iterate Iterator .................................................................................... 23

5 Validation of an Application Model using Constraints .............................................. 24

6 External Constraint Specification on an Application Model .................................... 28
  6.1 The Representation Part of a Profile .................................................................... 28
  6.2 The Constraints and Code Generator Part of a Profile ........................................ 29
  6.3 Loading a Profile .................................................................................................. 31

7 Transformations ......................................................................................................... 33

8 Code Generation using the Code Specifications ...................................................... 35

9 Runtime (OCL) Constraints ......................................................................................... 39
  9.1 Defining Runtime OCL Constraints .................................................................... 39
  9.2 Evaluating Runtime (OCL) Constraints .............................................................. 40
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Model Driven Architecture using profiles</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Modeling with OCL</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Architecture of the MDA Tool</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Inserting a model definition</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>The primitive types</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>The collection types</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>Simple class diagram</td>
<td>24</td>
</tr>
<tr>
<td>8</td>
<td>Evaluating the constraints associated with a class definition</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>Displaying constraints in the UML editor</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>Debugging an OCL constraint</td>
<td>26</td>
</tr>
<tr>
<td>11</td>
<td>The representation part of a profile</td>
<td>28</td>
</tr>
<tr>
<td>12</td>
<td>List of constraints and code generators in a profile</td>
<td>29</td>
</tr>
<tr>
<td>13</td>
<td>Example of an OCL invariant constraint</td>
<td>30</td>
</tr>
<tr>
<td>14</td>
<td>Example of an OCL operation definition</td>
<td>30</td>
</tr>
<tr>
<td>15</td>
<td>Replacing a profile</td>
<td>31</td>
</tr>
<tr>
<td>16</td>
<td>Validation after loading a profile</td>
<td>32</td>
</tr>
<tr>
<td>17</td>
<td>A simple model transformation</td>
<td>33</td>
</tr>
<tr>
<td>18</td>
<td>Defining a generation set</td>
<td>35</td>
</tr>
<tr>
<td>19</td>
<td>Generating code</td>
<td>35</td>
</tr>
<tr>
<td>20</td>
<td>Adding a runtime constraint</td>
<td>39</td>
</tr>
<tr>
<td>21</td>
<td>The object context of an OCL runtime constraint</td>
<td>40</td>
</tr>
<tr>
<td>22</td>
<td>Navigating through the constraints and generating code</td>
<td>41</td>
</tr>
<tr>
<td>23</td>
<td>The OCL meta-model (a part of)</td>
<td>42</td>
</tr>
<tr>
<td>24</td>
<td>Generative code for each OCL meta-model element</td>
<td>42</td>
</tr>
<tr>
<td>25</td>
<td>A writer to navigate to the child OCL model elements of an OCL if expression</td>
<td>43</td>
</tr>
</tbody>
</table>
1 Introduction

This document describes the specification and implementation of an UML (Unified Modeling Language) tool supporting Agile MDA (Model Driven Architecture). This introduction chapter gives some background of the UML, UML profiles and MDA.

1.1 UML Standardization through OMG

From version 1.5 onwards, the OMG\footnote{See also http://www.omg.org/ or more specifically http://www.uml.org/} (Object Management Group), responsible for the standardization of UML and MDA (Model Driven Architecture) introduced OCL (Object Constraint Language) and UML profiles. Both enrich the UML meta-model, but also the user defined application model.

The purpose of an UML profile is providing a customization mechanism to specialize a UML modeler such that it guides, checks and automates a UML development. An UML profile defines stereotypes, rules, constraints and code specifications. When applied to an UML model, it enriches that model and checks whether it conforms to the constraints and rules. It also may produce various project products, such as documents, test sequences, metrics, …etc.

An UML profile incorporates the know-how of users concerned about the development process for different application domains and the quality of the development work. An UML user may apply several UML profiles to one UML model or apply different UML profiles at different development phases. The idea of MDA is having specialist users to define profiles for specific domains and user to analyze, design and implement projects within those domains (see Figure 1).

![Figure 1: Model Driven Architecture using profiles](image)

1.2 Positioning the Object Constraint Language (OCL)

The Object Constraint Language can be used for three purposes (see Figure 2):

1. The Object Management Group (OMG) uses OCL to define constraints on the objects in the UML meta-model (see the top part of Figure 2). These constraints are defined in the UML superstructure specification and are part of the standard UML specification.

2. A domain specialist may define additional constraints on the objects in the UML meta-model. These constraints are defined in an UML profile (see the middle part of Figure 2). An example of such a constraint may be that multiple inheritances are not allowed within the domain for which to model.

3. While creating a model, an application user may define constraints on it. These constraints apply on the application model and not on the UML meta-model (see lower part of Figure 2).
The OMG also defines four levels of models:
1. M0: the actual user defined application, creating user objects.
2. M1: the application model, as shown in Figure 2, at the bottom right.
3. M2: the UML meta-model, as shown in Figure 2, at the top right.
4. M3: the UML meta-meta-model, defining a structure to store the UML meta-model and possibly also other meta-models.
2 The Architecture of the Agile MDA Tool

Figure 3 gives an overview of the architecture of the Agile MDA tool, consisting of three parts:

1. The **UML editor**. This is the main tool and offers a graphical interface to define an application model, to check its constraints, to transform a model into another, transform a model into code and to produce documents. To perform these tasks, the UML editor uses profiles defined by the other tools:
   - It uses an UML profile to display the required graphical representation of the different (stereotyped) objects, to check the constraints (interpreting the OCL constraints), to transform one model into another and to generate code.
   - It uses a document profile to produce the required documents from a given UML model.

2. An **interactive profile builder**. A profile contains domain specific OCL constraints, model definitions, model transformations, code generator(s), diagram specifications, graphical representations of classes and associations, stereotypes, …etc. To activate a profile, it must be loaded into a model, using the UML editor.

3. An **interactive document generator**. This generator uses a document (description) profile, in which the user defines the content of the required document and the order in which the data from the UML model should appear. To create such a document profile, there is an interactive document profiler, using the UML meta-model as its driver.

An UML profile may contain transformations and code generators. Both are expressed in OCL, extended with specific constructs to create UML model elements or to produce code. In fact, OCL offers powerful navigation mechanisms and a complete set of functions to handle collections. These can be used to transform UML associations and generalizations into sets of classes, which in turn can be used as drivers for transformations and code generations. The next chapter explains the different OCL extensions for transformations and code generations.
3 Transformations and Code Generations

Both transformations and code generations supported in the Agile MDA Tool are based on language extensions of OCL. These are rather small: a few context declarations, a new variable type and several predefined functions (in the OCL library).

3.1 Model Derivations with Transformations

A transformation transforms one model definition into another one. To create a new model definition, the user selects an existing model definition from which the new model definition will be derived and enters a name for new model definition. The profile editor then automatically inserts an empty transformation, as shown in Figure 4:

The default model definition does not contain a transformation, as this is the top model definition. The user now defines the required transformation actions, which will be performed in the UML editor when creating the required model. In general, the set of model definitions can be represented in a tree, as illustrated in Figure 4:

- Model definition M2 is derived from model definition M1, which in turn is derived from the default model definition.
- Model definition AltM1 is an alternative model definition, directly derived from the default model definition.

To create a transformed model, the UML editor first copies the data from the parent model and then performs the transformation on this copied model, which results in a modified model. When the parent model is also a model definition (as is the case for model definition M2 in Figure 4), the UML editor first creates this parent model by applying the corresponding transformation to it. The actual transformation is defined by OCL statements, which have been extended with creators, modifiers and removers of model elements (see section 3.3.1).
3.2 **Code Generators**

The purpose of applying a transformation is to modify the initial (default) model in such a way that it becomes easier to define a code generator. A code generator transforms a model into one or more text files. An obvious example of a code generator is the transformation of a model into a programming language, such as C++ or Pascal. However, other examples of code generators are script files containing instructions for testing, SQL files containing creation statements for a database structure and document files containing for instance an overview of requirements or constraints.

The complexity of the code generator depends on the model to transform, but also on the target language. For some code generators it may be necessary to transform the original model into a more appropriate model and for other no model transformations are necessary. So, any model definition can have one or more code generators associated to it (see also Figure 4, the command "Add Generator" at the right).

A code generator is equally defined by OCL statements, which have been extended with a text file type and a generator construct (see section 3.3.2).

3.3 **OCL Extensions**

The main use of OCL is to define constraints both on the meta-model and on the model (see section 1.2). However, as OCL is a powerful navigation language, simple extensions allow defining transformations and code generators, making it even more powerful.

This section uses an EBNF (Extended Backus Naur Formalism) notation to define the concrete syntax of these extensions. Where necessary, it uses the original syntax of the OCL specification and extends it.

The EBNF notation uses the following conventions:
- A production rule is a name between sharp brackets (< and >).
- Literal values (appearing in the concrete language) are written between single quotes (').
- A generic token is name.
- An part of a production rule place between square brackets ([ and ]) is optional.
- If a production rule contains choices, they are separated by a vertical bar symbol (|).

Example:

```plaintext
<ProductionRule> ::= 'production' Number [ ReferenceName ] | 'release' Name
```

This rule consists of two choices (the one starting with 'production' and the other with 'release'), separated by a bar symbol (|). The first choice contains an optional name (ReferenceName).

3.3.1 **Context Declarations for Transformations**

There are four types of context declarations for transformations. They are all specialized OCL "def" declarations:

1. A single "Transformation" definition automatically associated to a model definition (see section 3.1). It has no parameters and may call the other definitions to perform the transformation. Its return value is a Boolean type. The UML editor automatically calls a transformation when the user activates a model. A transformation may call transformational context definitions (see the next definitions) or any other ("def") definition.

2. A "Creator" definition, allowing the creation of UML model elements, such as packages, classes, associations, …etc. Every UML model element has a default creator, but the user may define several additional creators for a specific UML model element. A creator may have several parameters and must return the object it creates (Self). If there are several creators, the UML editor calls the creator with the matching actual parameters.
3. A "Modifier" definition, to modify an UML model element, by assigning values to its attributes and relations. So, a "modifier" definition may contain assignments, which is the only difference with a regular "def" definition.

4. A "Remover" definition to remove (free) an instance of an UML model element from the model. A remover has no parameters and must return a Boolean value. An UML model element may have only one user-defined remover.

As these context declarations are specialized OCL "def" constructs, their definitions fit into the OCL production rule <InvOrDefCS> and they both use the same syntax as a "def" construct:

```
<classifierContextDeclCS> ::= 'Context' Name <InvOrDefCS>
<invordefCS> ::= 'inv' [ Name ] ':' <OCLExpressionCS>
| 'Transformation' [ Name ] ':' <DefExpressionCS>
| 'Creator' [ Name ] ':' <DefExpressionCS>
| 'Modifier' [ Name ] ':' <DefExpressionCS>
| 'Remover' [ Name ] ':' <DefExpressionCS>
```

Example of a Transformation context declaration:
```
context Package transformation M1(): Boolean = True
```

This is the default (empty) definition of a transformation when defining a new model definition. The name for the transformation is the name of the model definition (see the model definition M1 in Figure 4). As a model definition is a specialized package UML model element, the context of a transformation is a package.

Example of a Creator context declaration
```
context Class creator NewClass(): Class = Self
```

As a creator or a remover is meant for a particular type of UML model element, the UML editor does not allow calling them by name. Instead, OCL has been extended with two predefined calls:

1. **NewInstance()**: OCLModelElement
   **NewInstance(Parameters)**: OCLModelElement
   Both must be applied to class type expression. The first format (without parameters) either calls a user-defined creator without parameters (if there is one) or the default creator, which is defined for each UML model element. The second format (with parameters) only calls the user-defined creator with the corresponding number and type of parameters.
   Examples of creator calls:
   ```
   Class.NewInstance()
   This calls the standard creator for the UML model element Class or a user-defined one if there is one without parameters.
   Class<<PlugClass>.NewInstance()
   This calls the standard creator for the UML model element Class, having a stereotype "PlugClass".
   ```

2. **FreeInstance()**: Boolean
   This either calls the user-defined remover (if there is one) or the default remover, which is defined for each UML model element.
   Example of remover call:
   ```
   cl.FreeInstance()
   This call removes the object cl, which is any kind of UML model element. It either calls the standard remover for that UML model element or the user-defined one, if it exists.
To assign a value to an attribute or relation, an OCL expression has been extended with an assign operator ("<-"), which can only be used in the context declarations for transformations. The assign operator is a binary operation returning a value of the same type of the attribute it is assigning to.

Example: An assignment to the attribute "name"

```
let s = name <- 'A'
```

The attribute "name" of a class has the type string. So, assignment also returns a string value.

### 3.3.2 Context Declarations for Code Generations

The code generator has two types of context declarations, both of which are specialized OCL "def" declarations:

1. A single "OutputControl" definition, which is called from the UML editor and has a single parameter defining the folder (directory) in which to put the generated code file(s). Its return value is a Boolean type.
2. One or several "OutputWriter" definitions, called from the "OutputControl" definition. The first parameter of an "OutputControl" is a "TextFile" type. An "OutputWrite" has a Boolean return value and may have more than one parameter.

As both context declarations are specialized OCL "def" constructs, their definitions fit into the OCL production rule <InvOrDefCS> and they both use the same syntax as a "def" construct:

```
<ClassifierContextDeclCS> ::= 'Context' Name <InvOrDefCS>

<InvOrDefCS> ::= 'inv' [ Name ] ':' <OCLExpressionCS>
| 'def' [ Name ] ':' <DefExpressionCS>
| 'OutputControl' [ Name ] ':' <DefExpressionCS>
| 'OutputWriter' [ Name ] ':' <DefExpressionCS>
```

Example of an OutputControl context declaration:

```
context Package <<plugGenerationSet>>
outputControl: GeneratePlugFiles(szDir: String): Boolean =
let Classes = class.AllInstances()
in classes -> iterate(cl; acc: Boolean = True
| acc and
let f1 = TextFile(szDir + '\Cl' + cl.name + '.pas')
in cl.OutputClFile(f1)
)
```

When the OutputControl context definition from the example above is called from the UML editor, it passes the name of the folder in which to generate the code file(s) in its szDir parameter. The body of the OutputControl context definition iterates through all the classes in the model using regular OCL statements. For each class, it defines a TextFile variable and calls an OutputWriter ("cl.OutputClFile") definition to generate the code into the text file.

Example of an OutputWriter context declaration:

```
context Class <<plugClass>>
outputWriter: OutputClFile(f: TexFile): Boolean =
f.Writeline('unit Cl' + cl.name + ';') +
f.Writeline('interface') +
f.Writeline('type') +
f.Writeline('  class Cl' + cl.name) +
...
f.Writeline('end;)') <> 0
```

The OutputWriter definition above consists of a simple series of straight calls to the predefined operation Writeline. Actual code generators are surely more complex than this simple example.
3.3.3 Variable Type "TextFile"

When defining a code generator, the OCL compiler supports the variable type `TextFile`. It can be used in a let statement and also as a parameter of other OCL operations (see section 3.3.2).

The declaration of a TextFile variable is very similar to a collection declaration and initialization. The EBNF definition from OCL have been extended as follows (the bold parts are the extensions):

```
<VariableDeclarationCS> ::= SimpleNameCS [':' <TypeCS>] ['=' <OclExpressionCS> ]

<TypeCS> ::= <PathNameCS>
  | <CollectionTypeCS>
  | <TupleTypeCS>
  | <TextFileType>

<TextFileType> ::= 'TextFile'

<OclExpressionCS> ::= <PropertyCallExpCS> | <VariableExpCS>
  | <OCLMessageCS>
  | <IfExpCS>
  | <LiteralExpCS>

<LiteralExpCS> ::= <EnumLiteralExpCS>
  | <CollectionLiteralExpCS>
  | <TupleLiteralExpCS>
  | <PrimitiveLiteralExpCS>
  | <TextFileLiteralExp>

<TextFileLiteralExp> ::= 'TextFile' '{' FileNameString '}'
```

The TextFile type is only available in code generation contexts (see section 3.3.2). Outside these contexts, the OCL compiler gives an error message when trying to use a TextFile variable.

An example of a text file variable definition is:

```
let f = TextFile{c:\UML\Test.pas} in ...
```

Remark that a backslash (\) is an escape character and has to be repeated if it is part of the string.

3.3.4 Predefined (Library) Operations

The OCL library specification has been extended with several predefined operations:

- Several operations are available on the new "TextFile" type, to write pieces of code to a text file.
- The string type has been extended with pattern matching operations and string search and replace operations.
- All primitive types have an extra operation to convert their value into a string.
- The type compatibility of some set operations have been extended to include the OrderedSet.

Chapter 4 (The OCL Library) describes these extensions in more detail.
4 The OCL Library

The OCL library contains the operations as specified in the OCL 2.0 specification, together with several additional or extended operations to offer generative actions or support manipulations or transformations. Deviations to the standard OCL have been marked as “(extension)”.

4.1 Operations on UML model elements

All UML model elements have the type OCL model element. This type supports the following operations:

- **OCLAsType(c: OCLType): OCLModelElement (extension)**
  Returns `Self`, recast to the type defined by `c`. This operation is only possible if `c` inherits from the current type of `Self`.
  The format of the `c` parameter may take the following forms:
  - A class name, as for instance class or classifier.
  - A class name, followed by a stereotype, as for instance class<<generationclass>>. This is an extension of the OCL library.
  - An expression having an OCLModelElement as a result, from which the class can be derived (as each UML model element has a class).

- **OCLIsTypeOf(c: OCLType): Boolean**
  Returns True, if `Self` is of the given type defined by `c`, which can take the same three formats as explained in OCLAsType.

- **OCLIsKindOf(c: OCLType): Boolean**
  Returns True, if `Self` conforms to the given type defined by `c`, which can take the same three formats as explained in OCLAsType. `Self` is conform to the type `c` it can be derived from that type.
  Example: if `Self` is a class, OCLIsKindOf(classifier) will return true.

- **AllInstances(): Set(OCLModelElement)**
  Returns all instances of `Self`, where `Self` must be a class name or an UML model element.
  Example: class.AllInstances() returns the set of all UML classes in the model.
  Example: class<<plugclass>>.AllInstances() returns the set of all UML classes in the model having the stereotype <<plugclass>>.

- **NewInstance():OCLModelElement (extension)**
  Calls a creator for the UML model element defined by self (see section 3.3.1).

- **FreeInstance():OCLModelElement (extension)**
  Calls a remover for the UML model element defined by self (see section 3.3.1).

4.2 Operations on Primitive Types

The OCL compiler and library support the primitive types shown in Figure 5:

![Diagram of primitive types](image)

Figure 5: The primitive types
### 4.2.1 Real Type Operations

- **Abs(): Real**
  Returns the absolute value of Self.

- **Floor(): Integer**
  Returns the largest integer, which is less than or equal to Self.

- **Round(): Integer**
  Returns the integer, which is closest to self. When there are two such integers, returns the largest one.

- **Max(r: Real): Real**
  Returns the maximum of Self and r.

- **Min(r: Real): Real**
  Returns the minimum of Self and r.

- **AsString(): String (extension)**
  Returns the string representation of Self.

The type real also has the following infix operations:

<table>
<thead>
<tr>
<th>Operator (extension)</th>
<th>Representation</th>
<th>Example</th>
<th>Priority (low to high)</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign (extension)</td>
<td>&lt;-</td>
<td>r1 &lt;- r2</td>
<td>1</td>
<td>Real</td>
</tr>
<tr>
<td>Equal</td>
<td>=</td>
<td>r1 = r2</td>
<td>2</td>
<td>Boolean</td>
</tr>
<tr>
<td>Not equal</td>
<td>&lt;&gt;</td>
<td>r1 &lt;&gt; r2</td>
<td>2</td>
<td>Boolean</td>
</tr>
<tr>
<td>Less</td>
<td>&lt;</td>
<td>r1 &lt; r2</td>
<td>2</td>
<td>Boolean</td>
</tr>
<tr>
<td>Less equal</td>
<td>&lt;=</td>
<td>r1 &lt;= r2</td>
<td>2</td>
<td>Boolean</td>
</tr>
<tr>
<td>Greater</td>
<td>&gt;</td>
<td>r1 &gt; r2</td>
<td>2</td>
<td>Boolean</td>
</tr>
<tr>
<td>Greater equal</td>
<td>&gt;=</td>
<td>r1 &gt;= r2</td>
<td>2</td>
<td>Boolean</td>
</tr>
<tr>
<td>Addition</td>
<td>+</td>
<td>r1 + r2</td>
<td>3</td>
<td>Real</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>r1 - r2</td>
<td>3</td>
<td>Real</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>r1 * r2</td>
<td>4</td>
<td>Real</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
<td>r1 / r2</td>
<td>4</td>
<td>Real</td>
</tr>
<tr>
<td>Unary minus</td>
<td>-</td>
<td>- r1</td>
<td>5</td>
<td>Real</td>
</tr>
</tbody>
</table>

As usual in other programming languages, the use of parentheses changes the operator priority.

### 4.2.2 Integer Type Operations

- **Abs(): Integer**
  Returns the absolute value of Self.

- **Div(i: Integer): Integer**
  Returns the number of times i (<> 0) fits completely within Self.

- **Mod(i: Integer): Integer**
  Returns the result of Self modulo i.

- **Max(i: Integer): Integer**
  Returns the maximum of Self and i.

- **Min(i: Integer): Integer**
  Returns the minimum of Self and i.

- **AsString(): String (extension)**
  Returns the string representation of Self.
The type integer has the following infix operations (overriding the operations from the real type):

<table>
<thead>
<tr>
<th>Operator</th>
<th>Representation</th>
<th>Example</th>
<th>Priority (low to high)</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign (extension)</td>
<td>&lt; -</td>
<td>i1 &lt;- i2</td>
<td>1</td>
<td>Integer</td>
</tr>
<tr>
<td>Addition</td>
<td>+</td>
<td>i1 + i2</td>
<td>2</td>
<td>Integer</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>i1 - i2</td>
<td>2</td>
<td>Integer</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>i1 * i2</td>
<td>3</td>
<td>Integer</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
<td>i1 / i2</td>
<td>3</td>
<td>Real</td>
</tr>
<tr>
<td>Unary minus</td>
<td>-</td>
<td>- i1</td>
<td>4</td>
<td>Integer</td>
</tr>
</tbody>
</table>

### 4.2.3 String Type Operations

String operations have been extended in two ways:
1. Some operations use patterns offering a much larger functionality than the standard OCL library.
2. Some operations have been added to support generative or transformation actions.

Before listing the string operations, the next section defines string patterns.

### 4.2.3.1 Pattern Definition

Some library string operations use patterns (regular expressions) for more sophisticated matching and replacing. A pattern consists of pattern elements and pattern operators:

- **Pattern elements** are shorthand definitions standing for one or more characters:
  - # defines any digit (0-9)
  - @ defines any alphabetical character (a-z, A-Z)
  - ? defines any character
  - [...] defines a general set of characters
  Examples of character sets are:
    - [01] defines a set of binary digits (0-1)
    - [@#] defines a set of alpha numerical characters (0-9, a-z, A-Z)
    - [#ABCDEF] defines a set of hexadecimal digits (0-9, A-F)
  Pattern elements may be concatenated: a pattern element followed by another pattern element forms a pattern that matches anything matched by the first, followed immediately by anything matched by the second.

- **Pattern operators** define operations on a pattern element:
  - ^ is the all but operator
    Examples are:
      - ^a defines all but the letter a
      - ^# defines the set of all characters except the digits
      - ^[@#] defines all but alphanumerical characters
      - ^[#ABCDEF] defines all but hexadecimal digits
  - * is the repetition operator, defining a sequence (zero or more) character sets:
    Examples are:
      - #* defines a string with 0 or more digits
      - [#ABCDEF]* defines a string with 0 or more hexadecimal digits
      - ^[#@]* defines a string with 0 or more non-alphanumeric characters
    When using the repetition operator in a string matching operation it always matches the longest string.
  - % is the optional operator (zero or one).
    Examples are:
      - #% defines a string 0 or 1 digit
      - [+-%] defines a string with 0 or 1 times + or -
  - \ is the deprive operator (to deprive the meaning of the special characters (#, @, ?, *, %, [, ,),
Note that special characters are only special in a given context:
- # is always special
- @ is always special
- ? is always special
- [ is only special outside character set definition
- ] is only special if a character set definition is specified
- ^ is only special outside character set definitions
- * is only special outside character set definitions
- % is only special outside character set definitions
- \ is always special

When two operators are used in sequence, they lose their special meaning:
Examples:
- ** matches 0 or more *
- a** matches 0 or more the letter a, followed by *
- ^^ matches all but ^
- \ matches \
- %* matches 0 or more %
- *% matches 0 or 1 times *

Examples of patterns are:
- Integer numbers
  - Pattern: ##*
  - Examples of valid integers: 0, 12345 and 11
- Pascal identifiers
  - Pattern: @[@#_]*
  - Examples of valid identifiers: a, abc and ab_1
- Real numbers
  - Pattern: [+-%]*%##*%eE*%[-+]*%##*
  - Examples of valid real numbers: 123.456, +1.23E4 and -0.56E-7
- Telephone numbers
  - Pattern: [+-%]*%##*%(%##*)%[#.]*
  - Example of a telephone number: +32(9)21.03.83
    The pattern above also matches the following string, which may not considered as valid:
    +0000032(9999999)…21…21…222222

4.2.3.2 String Operations using Patterns
These are operations using patterns as a parameter. All of them are extensions to the standard OCL library:
- **Matches(p: String): Boolean (extension)**
  Returns true if the string in Self completely matches the pattern in p.
- **StartsWith(p: String): Boolean (extension)**
  Returns true is the string in Self starts with the string defined by the pattern in p.
  Example: if Self contains **123String456**:  
  - StartsWith("#*") will return true.
  - StartsWith('123') also will return true.
  - StartsWith('123String456') is identical with Matches("?*") and will return true.
- **EndsWith(p: String): Boolean (extension)**
Returns true is the string in Self ends with the string defined by the pattern in p.
Example: if Self contains 123String456:
- EndsWith('#*') will return true.
- EndsWith('456') also will return true.

- **Includes(p: String): Boolean (extension)**
  Returns true, if the string in Self includes the string defined by the pattern in p.
  Example: if Self contains 123String456:
  - Includes('@*') will return true.
  - Includes('String') equally will return true.
  - Includes('**') is identical with Matches('**') and will return true.

- **SubstringMatch(p: String): String (extension)**
  Returns the sub string of the string in Self that matches the pattern in p.
  Example: if Self contains 123String456, SubstringMatch('@*') will return String.

- **SubstringBefore (p: String): String (extension)**
  Returns the part of the string in Self before the part that matches the pattern in p. If the matching part of the string starts at the first character in Self, the string returned is the empty string. If there is no match, returns the whole string in Self.
  Example: if Self contains 123String456, SubstringBefore('@*') will return 123.

- **SubstringAfter (p: String): String (extension)**
  Returns the part of the string in Self after the part that matches the pattern in p. If the matching part of the string ends at the last character in Self, the string returned is the empty string. If there is no match, returns the whole string in Self.
  Example: if Self contains 123String456, SubstringAfter('@*') will return 456.

- **Replace (mp: String, cs: String): String (extension)**
  Replaces all matching parts of the string in Self by the replacement control string cs. This replacement string may contain the following control characters:
  - *: stands for the matching part of the string to be replaced.
  - \: deprives the character * and itself of their special meanings.
  - All other characters are considered as replacement characters.
  Examples: if Self contains 1234567890:
  - Replace('#*', '[') will return [1234567890] (i.e. enclose the matching string part in brackets).
  - Replace('#*', '*,*') will return 1234567890,1234567890 (i.e. replace the matching string part by two of its occurrences, separated by a comma).
  - Replace('#*!', 'abc') will return abc (i.e. a pure substitution).
  - Replace('#!', ',*') will return * (i.e. replace the matching string part by the character *, as it has been deprived of its special meaning by the ',' character).
  - Replace('@*', '*') will not change the string, as the given pattern (@*) does not match any part of the string in Self.

### 4.2.3.3 String Operations based on Patterns

These operations use (internal) patterns to match strings. They have been added to increase the readability of OCL constraints.

- **MatchIdentifier(): Boolean (extension)**
  Returns true if the value in Self represents an identifier. An identifier starts with a letter, possibly followed by a sequence of letters, digits and the underscore character. This is identical to Matches('[@_#@]*').

- **MatchInteger(): Boolean (extension)**
  Returns true if the value in Self represents an integer number.
  This is identical to Matches('###*'), as an integer contains at least one digit.

- **MatchReal(): Boolean (extension)**
  Returns true if the value in Self represents a real number.
This is identical to Matches ("[+-]%##*.%#*[eE]%[+-]%##*").

- **MatchBoolean(): Boolean (extension)**
  Returns true if the string in Self either contains "0", "1", "true" (case insensitive) or "false" (case insensitive).

### 4.2.3.4 Other String Operations (including the standard operations)

- **Size(): Integer**
  Returns the number of characters in Self.

- **Concat(s: String): String**
  Returns the concatenation of Self and s. This is identical with the infix + operator: Self + s.

- **Substring(lower: Integer, upper: Integer): String**
  The substring of Self, starting at the position lower, up and including the position upper. Character positions run from 1 to Self.Size(). Raises an exception if either lower or upper is out of this string range.

- **ToReal(): Real**
  Converts Self to a real value, if the string in Self corresponds to a real value (Self.MatchReal() returns True). Raises an exception if the string in Self cannot be converted to a real value.

- **ToInteger(): Integer**
  Converts Self to an integer value, if the string in Self corresponds to an integer value (Self.MatchInteger() returns True). Raises an exception if the string in Self cannot be converted to an integer value.

- **ToBoolean(): Boolean (extension)**
  Converts the value in Self to a Boolean value or raises an exception. Only the values "0", "1", "true" (case insensitive) or "false" (case insensitive) can be converted to a Boolean value.

- **ToUpper (): String (extension)**
  Returns the uppercase value of Self.

- **ToLower (): String (extension)**
  Returns the lowercase value of Self.

- **Capitalize(): String (extension)**
  Returns the string in Self, with its first character capitalized.

- **Decapitalize(): String (extension)**
  Returns the string in Self, with its first character in lower case.

- **IndexOf(s: String): Integer (extension)**
  Returns the position of the string s in the string in Self or 0, if there is no match.

- **Trim(): String (extension)**
  Returns the string in Self, where all leading and trailing spaces have been removed.

- **NormalizeSpace(): String (extension)**
  Returns the string in Self, where all leading and trailing white spaces have been removed and where all internal sequences of white space are replaced by a single space. A white space is either a space or a tab.

The type string has the following infix operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Representation</th>
<th>Example</th>
<th>Priority (low to high)</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign (extension)</td>
<td>&lt;-</td>
<td>s1 &lt;- s2</td>
<td>1</td>
<td>String</td>
</tr>
<tr>
<td>Equal</td>
<td>=</td>
<td>s1 = s2</td>
<td>2</td>
<td>Boolean</td>
</tr>
<tr>
<td>Not equal</td>
<td>&lt;&gt;</td>
<td>s1 &lt;&gt; s2</td>
<td>2</td>
<td>Boolean</td>
</tr>
<tr>
<td>Less</td>
<td>&lt;</td>
<td>s1 &lt; s2</td>
<td>2</td>
<td>Boolean</td>
</tr>
<tr>
<td>Less equal</td>
<td>&lt;=</td>
<td>s1 &lt;= s2</td>
<td>2</td>
<td>Boolean</td>
</tr>
<tr>
<td>Greater</td>
<td>&gt;</td>
<td>s1 &gt; s2</td>
<td>2</td>
<td>Boolean</td>
</tr>
<tr>
<td>Greater equal</td>
<td>&gt;=</td>
<td>s1 &gt;= s2</td>
<td>2</td>
<td>Boolean</td>
</tr>
<tr>
<td>Catenation</td>
<td>+</td>
<td>s1 + s2</td>
<td>3</td>
<td>String</td>
</tr>
</tbody>
</table>
4.2.4 Boolean Type Operations

- **AsString(): String (extension)**
  Returns the string representation of *Self*, where *Self* is Boolean, Integer or Real.

The type Boolean has the following infix operations:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Interpretation</th>
<th>Priority (low to high)</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;- (extension)</td>
<td>b1 &lt;- b2</td>
<td>assigns b2 to b1</td>
<td>1</td>
<td>Boolean</td>
</tr>
<tr>
<td>implies</td>
<td>b1 implies b2</td>
<td>if b1 then b2 else true</td>
<td>2</td>
<td>Boolean</td>
</tr>
<tr>
<td>or</td>
<td>b1 or b2</td>
<td>if b1 then true else b2</td>
<td>3</td>
<td>Boolean</td>
</tr>
<tr>
<td>xor</td>
<td>b1 xor b2</td>
<td>(b1 or b2) and (b1 &lt;&gt; b2)</td>
<td>3</td>
<td>Boolean</td>
</tr>
<tr>
<td>and</td>
<td>b1 and b2</td>
<td>if b1 then b2 else false</td>
<td>3</td>
<td>Boolean</td>
</tr>
<tr>
<td>=</td>
<td>b1 = b2</td>
<td>true if b1 = b2</td>
<td>4</td>
<td>Boolean</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>b1 &lt;&gt; b2</td>
<td>true if b1 &lt;&gt; b2</td>
<td>4</td>
<td>Boolean</td>
</tr>
<tr>
<td>&lt;</td>
<td>b1 &lt; b2</td>
<td>true if b1 &lt; b2</td>
<td>4</td>
<td>Boolean</td>
</tr>
<tr>
<td>&lt;=</td>
<td>b1 &lt;= b2</td>
<td>true if b1 &lt;= b2</td>
<td>4</td>
<td>Boolean</td>
</tr>
<tr>
<td>&gt;</td>
<td>b1 &gt; b2</td>
<td>true if b1 &gt; b2</td>
<td>4</td>
<td>Boolean</td>
</tr>
<tr>
<td>&gt;=</td>
<td>b1 &gt;= b2</td>
<td>true if b1 &gt;= b2</td>
<td>4</td>
<td>Boolean</td>
</tr>
<tr>
<td>not</td>
<td>not b1</td>
<td>if b1 then false else true</td>
<td>5</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

4.2.5 TextFile Type Operations

The type TextFile is an extension (see section 3.3.3) used for code generation. It offers the following operations:

- **IndentInc(i: Integer): Integer (extension)**
  Increments the indentation counter and returns the new indentation value.

- **IndentDec(i: Integer): Integer (extension)**
  Decrements the indentation counter and returns the new indentation value.

- **NewLine(): Integer (extension)**
  Writes a new line on *Self* and starts the next line with the number of spaces, defined by the current value of the indentation counter. Only positive values of the indentation counter are taken into account. Returns the number of characters (bytes) written.

- **Writeline(s: String): Integer (extension)**
  Writes the string *s*, followed by a new line on *Self* and starts the next line with the number of spaces, defined by the current value of the indentation counter. Only positive values of the indentation counter are taken into account. Returns the number of characters (bytes) written. The string parameter may be an empty string.

- **Write(s: String): Integer (extension)**
  Write the string *s* on *Self* and returns the number of characters (bytes) written. The string parameter may be an empty string.

- **WriteSet(c: Collection, s: String): Integer (extension)**
  Writes the set elements in *c* on *Self* and returns the number of characters (bytes) written. The collection type stands for any kind of collection: set, orderedset, bag and sequence.

- **ConditionalWrite(c: <Condition>, s: String): Integer (extension)**
  Writes the string *s* on *Self*, if the condition *c* evaluates to True. This operation is equivalent to the OCL statement: if <Condition> then Self.write(s) else 0 endif

- **IsOpen(): Boolean (extension)**
  Returns true, if the file *Self* is open. Returns false, if the file *Self* could not be opened for writing.
4.3 Operations on Collection Types

The OCL compiler and library support the collection types shown in Figure 6:

![Figure 6: The collection types](image)

The following table gives an overview of collection operations and the type of collection (OrderedSet, Set, Sequence or Bag) to which it applies. The entries marked with "ext" are extensions to the type compatibility, compared to the OCL 2.0 specification.

<table>
<thead>
<tr>
<th>Operation name</th>
<th>Collection</th>
<th>OrderedSet</th>
<th>Set</th>
<th>Sequence</th>
<th>Bag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Count</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Includes</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Excludes</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>IncludesAll</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>ExcludesAll</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>IsEmpty</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>NotEmpty</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>= (Equal)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Union</td>
<td>√</td>
<td>(ext)</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Intersection</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Including</td>
<td>√ (ext)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Excluding</td>
<td>√ (ext)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>- (Difference)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>SymmetricDifference</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Flatten</td>
<td>√ (ext)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>AsOrderedSet</td>
<td>√ (ext)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>AsSet</td>
<td>√ (ext)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>AsSequence</td>
<td>√ (ext)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>AsBag</td>
<td>√ (ext)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Append</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Prepend</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>InsertAt</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>IndexOf</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>At</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>First</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Last</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>SubOrderedSet</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>SubSequence</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

The type "Collection" is an abstract OCL type from which the types "OrderedSet", "Set", "Sequence" and "Bag" are derived. It can only be used as a formal parameter type and can have any set type (including "Collection") as its actual parameter.

The type rules for the operation Union are:
The OCL Library

4.3.1 Collection Operations

These operations can be applied to all types of collections.

- **Size(): Integer**
  Returns the number of elements in the collection `Self`.

- **Includes(Object: T): Boolean**
  True, if `Object` is an element of the collection `Self`. False otherwise.

- **Excludes(Object: T): Boolean**
  True, if `Object` is not an element of the collection `Self`. False otherwise.

- **Count(Object: T): Integer**
  Returns the number of times the object `T` occurs in the collection `Self`.

- **IncludesAll(c: Collection(T)): Boolean**
  Returns True, if the collection `Self` contains all elements of the collection `c`. False otherwise.

- **ExcludesAll(c: Collection(T)): Boolean**
  Returns True, if the collection `Self` contains none of the elements of the collection `c`. False otherwise.

- **IsEmpty(): Boolean**
  True, if the collection `Self` is the empty collection. False otherwise.

- **NotEmpty(): Boolean**
  True, if the collection `Self` is not the empty collection. False otherwise.

4.3.2 OrderedSet Operations

- **Union(s: OrderedSet(T)): OrderedSet(T) (extension)**
  Returns the union of the ordered set `Self` and the ordered set `s`, giving a new ordered set.

- **Union(s: Set(T)): Set(T) (extension)**
  Returns the union of the ordered set `Self` and the set `s`, giving a new set.

- **Union(s: Sequence(T)): Sequence(T) (extension)**
  Returns the union of the ordered set `Self` and the sequence `s`, giving a new sequence.

- **Union(s: Bag(T)): Bag(T) (extension)**
  Returns the union of the ordered set `Self` and the bag `s`, giving a new bag.

- **Including(Object: T): OrderedSet(T) (extension)**
  Returns the ordered set containing the elements of the ordered set `Self`, followed by the object `T`. This operation is identical to the operation Append (see further).

- **Excluding(Object: T): OrderedSet(T) (extension)**
  Returns the ordered set containing the elements of the ordered set `Self` without the object `T`. For the operations "Including", "Excluding" and "Flatten", the type of the result is identical to the type of the set to which the operation has been applied.
• **Flatten(): OrderedSet(T) (extension)**
  If the elements of the ordered set Self are collections, this returns a set containing the elements of these collections, removing duplicate elements. Otherwise returns the ordered set Self.

• **AsSet(): Set(T) (extension)**
  Returns a set containing all elements of the ordered set Self.

• **AsOrderedSet(): OrderedSet(T) (extension)**
  Returns the ordered set Self.

• **AsSequence(): Sequence(T) (extension)**
  Returns a sequence containing all elements from the ordered set Self in an undefined order.

• **AsBag(): Bag(T) (extension)**
  Returns a bag containing all elements from the ordered set Self.

• **Append(Object : T): OrderedSet(T)**
  Returns the ordered set with the elements of the ordered set Self, followed by the object T.

• **Prepend(Object : T): OrderedSet(T)**
  Returns the ordered set containing the object T, followed by the elements of the ordered set Self.

• **InsertAt(Index: Integer, Object : T): OrderedSet(T)**
  Returns the ordered set containing the elements of the ordered set Self, with the object T inserted at position Index. The first element of an ordered set is at position 1. Raises an exception if the Index is out of the current bounds of the ordered set Self.

• **At(Index: Integer): T**
  Returns the element at the position Index from the ordered set Self. The first element of an ordered set is at position 1. Raises an exception if the Index is out of the current bounds of the ordered set Self.

• **IndexOf(Object: T): Integer**
  Returns the position of the object T in the ordered set Self or zero (0) if Object T does not belong to the ordered set Self. The first element of an ordered set is at position 1.

• **First(): T**
  Returns the first element of the ordered set Self.

• **Last(): T**
  Returns the last element of the ordered set Self.

• **SubOrderedSet(Lower: Integer, Upper: Integer): OrderedSet(T)**
  Returns the sub set (an ordered set) containing the elements of the ordered set Self, starting at the position lower, up to and including the position upper. The elements in an ordered set are numbered, starting from 1. Raises an exception if either the index Lower or the index Upper is out of the current bounds of the ordered set Self.

The type OrderedSet also has one infix operation:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Representation</th>
<th>Example</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equality</td>
<td>=</td>
<td>o1 = o2</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

### 4.3.3 Set Operations

• **Union(s: Set(T)): Set(T)**
  Returns the union of the set Self and the set s, giving a new set.

• **Union(s: OrderedSet(T)): Set(T) (extension)**
  Returns the union of the set Self and the ordered set s, giving a new set.

• **Union(s: Sequence(T)): Sequence(T)**
  Returns the union of the set Self and the sequence s, giving a new sequence.

• **Union(s: Bag(T)): Bag(T)**
  Returns the union of the set Self and the bag s, giving a new bag.
The OCL Library

- **Intersection(s: Set(T)): Set(T)**
  Returns the intersection of the set `Self` with the set `s`, giving a new set containing the elements that are both in `self` and `s`.

- **Intersection(s: Bag(T)): Set(T)**
  Returns the intersection of the set `Self` with the bag `s`, giving a new set containing the elements that are both in `self` and `s`.

- **Including(Object: T): Set(T)**
  Returns the set containing the elements of the set `Self` and the object `T`.

- **Excluding(Object: T): Set(T)**
  Returns the set containing the elements of the set `Self` without the object `T`.

- **SymmetricDifference(s: Set(T)): Set(T)**
  Returns the set containing the elements that are in the set `Self` or the set `s`, but not in both.

- **Flatten(): Set(T)**
  If the elements of the set `Self` are collections, this returns a set containing the elements of these collections. Otherwise returns the set `Self`.

- **AsSet(): Set(T)**
  Returns the set `Self`.

- **AsOrderedSet(): OrderedSet(T)**
  Returns an ordered set containing all elements from the set `Self` in an undefined order.

- **AsSequence(): Sequence(T)**
  Returns a sequence containing all elements from the set `Self` in an undefined order.

- **AsBag(): Bag(T)**
  Returns a bag containing all elements from the set `Self`.

The type `Set` also has two infix operations:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Representation</th>
<th>Example</th>
<th>Priority (low to high)</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equality</td>
<td>=</td>
<td>s1 = s2</td>
<td>1</td>
<td>Boolean</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>s1 – s2</td>
<td>2</td>
<td>Set</td>
</tr>
</tbody>
</table>

### 4.3.4 Sequence Operations

- **Union(s: OrderedSet(T)): Sequence(T) (extension)**
  Returns the union of the sequence `Self` and the ordered set `s`, giving a new sequence.

- **Union(s: Set(T)): Sequence(T)**
  Returns the union of the sequence `Self` and the set `s`, giving a new sequence.

- **Union(s: Sequence(T)): Sequence(T)**
  Returns the sequence `Self`.

- **Union(s: Bag(T)): Bag(T)**
  Returns the union of the sequence `Self` and the bag `s`, giving a new bag.

- **Including(Object: T): Sequence(T) (extension)**
  Returns the sequence containing the elements of the sequence `Self` followed by the object `T`. This operation is identical to the operation Append (see further).

- **Excluding(Object: T): Sequence(T) (extension)**
  Returns the sequence containing the elements of the sequence `Self` without the object `T`.

- **Flatten(): OrderedSet(T)**
  If the elements of the sequence `Self` are collections, this returns a sequence containing the elements of these collections. Otherwise returns the sequence `Self`.

- **AsSet(): Set(T)**
  Returns a set containing all elements of the sequence `Self`, removing duplicate elements.

- **AsOrderedSet(): OrderedSet(T) (extension)**
  Returns an ordered set containing all elements of the sequence `Self`, removing duplicate
elements.

- **AsSequence(): Sequence(T)** *(extension)*
  Returns the sequence *Self*.

- **AsBag(): Bag(T)*
  Returns a bag containing all elements from the sequence *Self*.

- **Append(Object : T): Sequence (T)**
  Returns the sequence with the elements of the sequence *Self*, followed by the object T.

- **Prepend(Object : T): Sequence (T)**
  Returns the sequence containing the object T, followed by the elements of the sequence *Self*.

- **InsertAt(Index: Integer, Object : T): Sequence (T)**
  Returns the sequence containing the elements of the sequence *Self*, with the object T inserted at position Index. The first element of a sequence is at position 1. Raises an exception if the Index is out of the current bounds of the sequence *Self*.

- **At(Index: Integer): T**
  Returns the element at the position Index from the sequence *Self*. The first element of a sequence is at position 1. Raises an exception if the Index is out of the current bounds of the sequence *Self*.

- **IndexOf(Object: T): Integer**
  Returns the position of the object T in the sequence *Self* or zero (0) if Object T does not belong to the sequence *Self*. The first element of a sequence is at position 1.

- **First(): T**
  Returns the first element of the sequence *Self*.

- **Last(): T**
  Returns the last element of the sequence *Self*.

- **SubSequence(Lower: Integer, Upper: Integer): Sequence(T)**
  Returns the subset (a sequence) containing the elements of the sequence *Self*, starting at the position lower, up to and including the position upper. The elements in a sequence are numbered, starting from 1. Raises an exception if either the index Lower or the index Upper is out of the current bounds of the sequence *Self*.

The type Sequence also has one infix operation:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Representation</th>
<th>Example</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equality</td>
<td>=</td>
<td>s1 = s2</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

### 4.3.5 Bag Operations

- **Union(s: Set(T)): Bag(T)**
  Returns the union of the bag *Self* and the set s, giving a new bag.

- **Union(s: OrderedSet(T)): Bag(T)** *(extension)*
  Returns the union of the bag *Self* and the ordered set s, giving a new bag.

- **Union(s: Sequence(T)): Bag(T)**
  Returns the union of the bag *Self* and the sequence s, giving a new bag.

- **Union(s: Bag(T)): Bag(T)**
  Returns the union of the bag *Self* and the bag s, giving a new bag.

- **Intersection(s: Set(T)): Set(T)**
  Returns the intersection of the bag *Self* with the set s, giving a new set containing the elements that are both in *Self* and s.

- **Intersection(s: Bag(T)): Bag(T)**
  Returns the intersection of the bag *Self* with the bag s, giving a new bag containing the elements that are both in *Self* and s.
The OCL Library

- **Including(Object: T): Bag(T)**
  Returns the bag containing the elements of the bag \( \text{Self} \) and the object \( T \).
- **Excluding(Object: T): Bag(T)**
  Returns the bag containing the elements of the bag \( \text{Self} \) without the object \( T \).
- **Flatten(): Bag(T)**
  If the elements of the bag \( \text{Self} \) are collections, this returns a bag containing the elements of these collections. Otherwise returns the bag \( \text{Self} \).
- **AsSet(): Set(T)**
  Returns a set containing all the elements of the bag \( \text{Self} \), removing the duplicate elements.
- **AsOrderedSet(): OrderedSet(T)**
  Returns an ordered set containing all elements from the bag \( \text{Self} \) in an undefined order. Duplicate elements are removed.
- **AsSequence(): Sequence(T)**
  Returns a sequence containing all elements from the bag \( \text{Self} \) in an undefined order.
- **AsBag(): Bag(T)**
  Returns the bag \( \text{Self} \).

The type bag also has an infix operation:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Representation</th>
<th>Example</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equality</td>
<td>=</td>
<td>( b1 = b2 )</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

4.4 **Iterators**

The following table gives an overview of iterators and the type of collection (OrderedSet, Set, Sequence or Bag) to which it applies. The entries marked with "ext" are extensions to the type compatibility, compared to the OCL 2.0 specification.

<table>
<thead>
<tr>
<th>Iterator name</th>
<th>Collection</th>
<th>OrderedSet</th>
<th>Set</th>
<th>Sequence</th>
<th>Bag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exists</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Forall</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>IsUnique</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Any</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>One</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Collect</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Select</td>
<td>√ (ext)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Reject</td>
<td>√ (ext)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>CollectNested</td>
<td>√ (ext)</td>
<td>√ (ext)</td>
<td>√</td>
<td>√ (ext)</td>
<td>√ (ext)</td>
</tr>
<tr>
<td>SortedBy</td>
<td>√ (ext)</td>
<td>√ (ext)</td>
<td>√</td>
<td>√ (ext)</td>
<td>√ (ext)</td>
</tr>
</tbody>
</table>

The type rules for SortedBy are:
- \( \text{OrderedSet} \) -> \( \text{SortedBy()} \) \( \rightarrow \) \( \text{OrderedSet} \)
- \( \text{Set} \) -> \( \text{SortedBy()} \) \( \rightarrow \) \( \text{OrderedSet} \)
- \( \text{Sequence} \) -> \( \text{SortedBy()} \) \( \rightarrow \) \( \text{Sequence} \)
- \( \text{Bag} \) -> \( \text{SortedBy()} \) \( \rightarrow \) \( \text{Sequence} \)

The general format of an iterator is: "\( \text{Source} \) -> \( \text{Name(Iterators | Body)} \)\), where "Iterators" are local variables used to traverse the collection "\( \text{Source} \)" and "\( \text{Body} \)" is a filter expression to match the elements from the collection "\( \text{Source} \)".

4.4.1 **Collection Iterators**

The following iterators are applicable on all collection types:
- \( \text{Source} \) -> \( \text{Exists(Iterators | Body): Boolean} \)
  Returns true, if \( \text{Body} \) evaluates true for at least one element in the collection \( \text{Source} \).
Otherwise returns false.

- **Source -> ForAll(Iterators | Body): Boolean**
  Returns true, if Body evaluates true for all elements in the collection Source. Otherwise returns false.

- **Source -> IsUnique(Iterator | Body): Boolean**
  Returns true, if Body evaluates to a different value for all elements in the collection Source. Otherwise returns false. IsUnique may use at most one iterator variable.

- **Source -> Any(Iterator | Body): T**
  Returns an element from the collection Source for which Body evaluates true. If the collection Source contains several elements for which Body evaluates true, only one is returned. Raises an exception if there are no elements for with Body evaluates true. Any may use at most one iterator variable.

- **Source -> One(Iterator | Body): Boolean**
  Returns true, if there is exactly one element from the collection Source for which Body evaluates true. False otherwise. One may use at most one iterator variable.

- **Source -> Collect(Iterator | Body): Bag(T)**
  Returns a bag, consisting of elements created by applying Body to each element of the collection Source. Collect may use at most one iterator variable.

Example: Suppose the collection source contains elements having an attribute name. The next iterator creates a bag containing the names of these elements:

```
Source -> Collect(r | r.name)
```

### 4.4.2 OrderedSet Iterators

- **Source -> Select(Iterators | Body): OrderedSet (extension)**
  Returns the sub set (an ordered set) of the ordered set Source for which Body evaluates true.

- **Source -> Reject(Iterators | Body): OrderedSet (extension)**
  Returns the sub set (an ordered set) of the ordered set Source for which Body evaluates false.

- **Source -> CollectNested(Iterators | Body): Bag (extension)**
  This iteration is identical to Collect, as the OCL runtime immediately flattens the resulting bag.

- **Source -> SortedBy(Iterators | Body): OrderedSet (extension)**
  Returns the ordered set containing all elements of the ordered set Source. The element for which Body has the lowest value comes first, and so on. The type of Body must have the "<" operation defined. The "<" operation must return a Boolean value and must be transitive, i.e. if a < b and b < c then a < c. SortedBy may use at most one iterator variable.

### 4.4.3 Set Iterators

- **Source -> Select(Iterators | Body): Set**
  Returns the sub set (a set) of the set Source for which Body evaluates true.

- **Source -> Reject(Iterators | Body): Set**
  Returns the sub set (a set) of the set Source for which Body evaluates false.

- **Source -> CollectNested(Iterators | Body): Bag (extension)**
  This iteration is identical to Collect, as the OCL runtime immediately flattens the resulting bag.

- **Source -> SortedBy(Iterators | Body): OrderedSet (extension)**
  Returns the ordered set containing all elements of the set Source. The element for which Body has the lowest value comes first, and so on. The type of Body must have the "<" operation defined. The "<" operation must return a Boolean value and must be transitive, i.e. if a < b and b < c then a < c. SortedBy may use at most one iterator variable.
4.4.4 Sequence Iterators

- **Source -> Select(Iterators | Body): Sequence**
  Returns the sub set (a sequence) of the sequence `Source` for which Body evaluates true.

- **Source -> Reject(Iterators | Body): Sequence**
  Returns the sub set (a sequence) of the sequence `Source` for which Body evaluates false.

- **Source -> CollectNested(Iterators | Body): Bag (extension)**
  This iteration is identical to Collect, as the OCL runtime immediately flattens the resulting bag.

- **Source -> SortedBy(Iterators | Body): Sequence**
  Returns the sequence containing all elements of the sequence `Source`. The element for which Body has the lowest value comes first, and so on. The type of Body must have the "<" operation defined. The "<" operation must return a Boolean value and must be transitive, i.e. if a < b and b < c then a < c. SortedBy may use at most one iterator variable.

4.4.5 Bag Iterators

- **Source -> Select(Iterators | Body): Bag**
  Returns the sub set (a bag) of the bag `Source` for which Body evaluates true.

- **Source -> Reject(Iterators | Body): Bag**
  Returns the sub set (a bag) of the bag `Source` for which Body evaluates false.

- **Source -> CollectNested(Iterators | Body): Bag (extension)**
  This iteration is identical to Collect, as the OCL runtime immediately flattens the resulting bag.

- **Source -> SortedBy(Iterators | Body): Sequence**
  Returns the sequence containing all elements of the bag `Source`. The element for which Body has the lowest value comes first, and so on. The type of Body must have the "<" operation defined. The "<" operation must return a Boolean value and must be transitive, i.e. if a < b and b < c then a < c. SortedBy may use at most one iterator variable.

4.4.6 The Iterate Iterator

The Iterate iterator is a generic iterator allowing to create new iterators or to describe the existing operators. An accumulation builds one value while iterating over a collection.

The general form of the iterate iterator is:

```
Collection -> Iterate(iterators;
    accumulator: type = initialization |
    Body with elem and accumulator)
```

The accumulator gets an initial value in its definition. While iterating over the collection, the body is evaluated for the given iterators. The result of this evaluation is then assigned (added) to the accumulator. This way, the value of the accumulator is built up during the iteration over the collection.

Example: the collect iterator can be expressed in terms of the iterate iterator.

"Source -> collect(x | x.property)" is identical with

"Source -> iterate(x; acc = bag{} | acc -> including(x.property))"
Validation of an Application Model using Constraints

To illustrate how OCL constraints operate, consider a simple class diagram, shown in Figure 7:

Class B inherits from class A, class C inherits from B and class A inherits from class C. This is a circular inheritance, which is not allowed in UML. So, when validating this model, the UML editor should give at least one error message.

As this is an error condition that applies to classes, the OCL constraint to check it must be attached to a class type (or its parent type, called Classifier).

The OCL constraint checking this condition is:

\[
\text{context Classifier}
\]
\[
\text{inv: not (self.AllParents(Set{}) -> includes (self))}
\]

The operation "AllParents" builds the set of all direct and indirect parents of a classifier and checks if the classifier belongs to that set ("includes"). If so, the model contains an error and the constraint fires.

Clearly, this constraint does not work without the companion operation "AllParents":

\[
\text{context: Classifier}
\]
\[
\text{def: AllParents(in callerParents: set(Classifier)):
  let parents = general, 
  gen = callerParents -> union(parents) 
  in parents ->
  union(parents ->
  collect(p | if callerParents -> includes(p) 
  then set()
  else 
  p.AllParents(gen)
  endif 
  ) ->
  AsSet()}
\]

This operation introduces two variables: "parents" contains the direct parents of the classifier for which this operation has been called and "gen" contains the union of the direct parents and the set of classifiers passed as a parameter to this operation.

The operation then iterates through the list of direct parents to find their parents by a recursive call. This recursion stops if there are no direct parents (set in "parents" is empty, so there is no iteration) or when the set passed as a parameter already contains the class found in the direct parents (if statement in the collect predefined operation).

When entering the class diagram of Figure 7 into the UML editor and performing the validation command, the UML editor will evaluate all constraints associated to all UML elements in the UML model and display messages for all constraints that were violated in the model. In this example, the UML editor will display three validation errors, as shown in Figure 8.
Validation of an Application Model using Constraints

When clicking on a constraint error, the UML editor will open the definition of the class containing the error. The definition of class A in the example of Figure 8 (see the window in the upper right corner) is a class without stereotype and without attributes or operations.

This class definition window contains three tabs: one with the class definition, one with user comments (possibly explaining the purpose of that class) and one with OCL constraints. Clicking on this latter tab lists the constraints associated with the class definitions. Double clicking on a constraint gives its definition, as shown in Figure 9.
A constraint definition consists of the following parts (see the right part of Figure 9):

- Its name. This name appears at the beginning of a validation error message.
- Its error level. The UML editor offers three levels (error, warning and hint). When performing a global validation, the user may decide only to see the errors and hide the other messages.
- The message to display when a constraint is violated.
- The OCL definition of the constraint.

When validating a large UML model, many constraint errors may result. The user can limit this number by setting an upper limit on the number of errors to display or by hiding the warning and hint messages. However, it is also possible to use a limited validation on an UML element:

- When selecting the "Validate" command in a definition window (see the window in the upper right corner of Figure 8), only the constraints for that UML element are validated.
- When selecting a constraint first and then selecting the "Validate" command in a definition window, the UML editor only validates this single constraint.

Although meant for the expert (domain) user, the UML editor also allows debugging an OCL constraint. When the user selects a constraint and clicks the "Debug" command (see the window in the upper right corner of Figure 8), a debugger window comes up as shown in Figure 10:

- The top window shows the OCL texts that are currently evaluated. The bar at the left shows the executable OCL statements (blue dots) and the arrow shows the currently executed line.
- The variables window at the left shows the variables in effect, together with their type, between brackets. Selecting one of them shows all its details in the window at the right.
- The call stack window at the left shows the list of operations currently in execution, together
with their context name between parentheses. Clicking on an element of the call stack displays the corresponding OCL source and variables.

- The sources window shows the list of sources together with their context name between parentheses. These sources are the ones needed for the currently evaluated OCL expression. Initially, this window shows the sources that could be detected statically. However, some sources can only be determined at runtime. These sources only appear in this window the first time they are called.

Clicking on an element of the sources list displays the corresponding OCL source and breakpoints. This allows a user to set breakpoints without having to wait until an OCL source is executed.

- The small window at the bottom show the result of the constraint evaluation (True or False) or show a runtime error message when an exception has occurred during the evaluation.

The action bar at the top of the debugger window contains the debugger commands:
- The **Run** command executes the OCL constraint up to the next breakpoint or until the end of the constraint has been reached.
- The **Step** command executes the next OCL line, marked with a blue dot.
- The **Breakpoint** command marks a line with a breakpoint (showed as a red dot). When executing an OCL constraint it stops when reaching that breakpoint.
- The **Clear Breakpoints** command clears all breakpoints set in the different OCL sources.
- The **Reset** command restarts the execution of the OCL constraint and resets all variables.
- The **Exit** command leaves the debugger.
6 External Constraint Specification on an Application Model

An UML editor is a generic tool to develop and investigate a model. When using it in the context of a domain, one may define a set of informal rules and conventions to express or model domain specific knowledge and information. However, the OMG has introduced UML profiles to enrich an UML editor with domain specific knowledge, conventions and constraints.

The interactive profile editor offers a user interface to define and edit UML profiles. It contains a part to define the elements needed within a given domain and another part to specify constraints and a code generator for that domain. The next sections describe both parts in more detail.

6.1 The Representation Part of a Profile

Figure 11 shows the first tab of the profile editor, in which the expert domain user can define the UML elements that apply within a given domain:

Figure 11: The representation part of a profile

- **Exposed elements** are the UML elements that may be used within the domain. These elements may be chosen from the list of all available UML elements, which the profile editor gets from the UML meta-model. An UML element may or may not have a specific stereotype. This stereotype is a user defined name and refines the generic UML elements.

- **Diagram definitions** are the UML diagrams that are supported within the domain. If a model contains other diagrams than defined in a profile, they are hidden when that profile is loaded. There are two types of diagrams:
  - Predefined ones: to allow a particular diagram in a model, simply check it.
  - User defined ones: by clicking the Add button, the user may define a new diagram, even diagrams that are not defined by OMG.

- **Shape objects** allow defining the shape of different UML elements appearing in a diagram. Equal UML elements, but with a different stereotype may be represented differently. This greatly enhances the readability of diagrams. The profile editor offers a set of basic shapes (rectangle, ellipses, diamonds, …etc), lines and colors that may be combined to any kind of representation of an UML element.
• **Data types** lists the data types that may be used e.g. for attributes when creating a model within the domain. The data types to use must be checked.

• **Stereotypes** define the stereotypes that may be used in the model. Each stereotype also may define additional tag (extra UML attributes) associated with that stereotyped element. The UML editor recognizes such tags and allows the user to enter values for them when creating an element with that stereotype.

• **Connection types** allow defining the representation of different UML relations, such as associations and generalizations. As for the UML elements, the profile editor offers a list of line types, arrow types, … etc.

### 6.2 The Constraints and Code Generator Part of a Profile

Figure 12 shows the list of constraints and code generation parts of a profile:

![List of constraints and code generators in a profile](image)

- **A constraint** is an invariant that applies in the context of an UML element. Such an element may or may not have a stereotype. A constraint adds to the default constraints of the UML editor. As the default constraints, a domain constraint may have a level (error, warning or hint). The UML editor prevents the call to a generator when the validation detects constraint errors. Figure 13 gives an example of a complete definition of an OCL constraint. The part at the right shows the UML model element to which the constraint applies, its attributes, its relations and its operations. The suffix "r", "w" or "rw" after each property indicates whether this property can only be read, can only be written to (i.e. give it a value through an assign operator) or can be read and written to.
An **operation** can be called from constraints to perform more elaborate navigation or checking actions. Operations may have parameters and return a value and may be recursive. Figure 14 gives an example of an OCL operation definition, which can be called from a constraint, a generator or a writer. Operation mainly gathers information by navigating through an UML model.
• A **transformation** is a definition (see section 3.3.1), defining a transformation between two model definitions. The profile editor automatically inserts an empty transformation between two model definitions.

• A **creator** is a definition to create an UML model element as part of a transformation (see section 3.3.1). Each UML model element has a default creator (not shown in the profile editor), but may have one or several user-defined creators. A creator is called through a predefined operation *NewInstance* (see the sections 3.3.1 and 4.1).

• A **modifier** is a definition to modify UML model elements through assignments, as part of a transformation (see section 3.3.1).

• A **remover** is a definition to remove an UML model element as part of a transformation (see section 3.3.1). Each UML model element has a default remover (not shown in the profile editor), but may have a user-defined one. A remover is called through a predefined operation *FreeInstance* without parameters (see the sections 3.3.1 and 4.1).

• A **generator** is an "OutputControl" definition (see section 3.3.2), defining a code generator. This is a specialized operation, directly called from the UML editor. It only can have one string parameter, containing the folder (directory) on which to place the generated code file(s) and returns a Boolean value, where True means a successful generation. A profile only contains one generator.

• A **writer** is an "OutputWriter" definition (see section 3.3.2), which is a specialized operation producing code. Its may have several parameters. However, its first parameter must be of the type "TextFile" (see section 3.3.3).

Constraints, operations, transformations and code generators are written in OCL. Only transformation definitions (transformation, creator, modifier and remover) and code generating definitions (generator and writers) may use the OCL extensions as described in chapter 3. The current profile editor accepts the complete OCL language, but does not support pre and post context definitions and signaling.

### 6.3 Loading a Profile

A completed profile can be loaded with an existing UML model or installed when creating a new UML model. Figure 15 shows the window of the UML editor to replace the default profile by a new profile called "Plugable Objects", while preserving the default constraints.

![Figure 15: Replacing a profile](image-url)

When replacing the default profile by the "Plugable Objects" profile into the UML model with the simple class diagram of Figure 7 and performing the validation again, now gives more messages, as shown in Figure 16. These messages come from the constraints defined in the profile. In this case, the messages are warnings, which would not prevent the code generator to produce code.
Figure 16: Validation after loading a profile
# Transformations

When defining a new model definition in the profile editor, all constraints and operations from the parent model are still available. So, users do not have to replicate the OCL code from one model definition to another.

In most cases, the OCL code associated to a particular model definition is limited to:

- The transformation, containing the OCL code modifying the parent model definition into the new model definition. This transformation may call creators, modifiers and removers. An example of such transformation may be the replacement of n-m associations, as shown in Figure 17.

![Figure 17: A simple model transformation](image)

The OCL code to perform this transformation is as follows:

```ocl
class Package
context Package
transformation PIM_2(): Boolean =
let
    classes = class.AllInstances()->AsOrderedSet(),
    assocs = association.AllInstances()->AsOrderedSet()
in
    assocs->iterate(assoc; b: Boolean = True
        if assoc.ownedEnd->size() = 2
            then
                assoc.SplitNMAssociation() and
                assoc.FreeInstance()
            else
                true
        endif or b)

context Association
modifier: SplitNMAssociation(): Boolean =
let
    end1 = ownedEnd->First(),
    end2 = ownedEnd->Last(),
    class1 = end1.type,
    class2 = end2.type
in
    if ((end1.upper > 1) or (end1.upper = -1)) and
        ((end2.upper > 1) or (end2.upper = -1))
    then
        let
            newClass = Class.NewInstance(), -- use default creator
            ncl1 = newClass.owner <- owner,
            nc2 = newClass.name <- 'Assoc_' + name,
```
Transformations

newAssoc1 = Association.NewInstance(), -- use default creator
na11 = newAssoc1.owner <- owner,
na12 = newAssoc1.name <- name + '_1',
newAssoc2 = Association.NewInstance(), -- use default creator
na21 = newAssoc2.owner <- owner,
na22 = newAssoc2.name <- name + '_2',
newEnd1 = AssociationEnd.NewInstance(), -- use default creator
ne11 = newEnd1.owner <- newAssoc1,
ne12 = newEnd1.type <- newClass,
ne13 = newEnd1.lower <- end2.lower,
ne14 = newEnd1.upper <- end2.upper,
newEnd2 = AssociationEnd.NewInstance(), -- use default creator
ne21 = newEnd2.owner <- newAssoc2,
ne22 = newEnd2.type <- newClass,
ne23 = newEnd2.lower <- end1.lower,
ne24 = newEnd2.upper <- end1.upper,

e11 = end1.owner <- newAssoc1,
e12 = end1.lower <- 1,
e13 = end1.upper <- 1,
e21 = end2.owner <- newAssoc2,
e22 = end2.lower <- 1,
e23 = end2.upper <- 1

in True
else True
Endif

After the execution of this transformation, the model no longer contains n-m-associations, which may reduce the complexity of a code generator.

- Specific constraints or operations for new UML model elements being introduced by the transformation. An example may be a constraint on the upper limit on the number of associated elements.
- One or more code generators to apply to the given model definition.
8 Code Generation using the Code Specifications

To keep the complexity of a large UML model under control, a user may use a hierarchy of packages. Each package may in turn contain several UML elements (e.g. classes) and diagrams. When generating code for an application, it may be necessary to select a subset of the model for which to generate code.

The UML editor considers a package having the stereotype "GenerationSet" as a container of packages and classes for which to generate code. Figure 18 shows the definition of a generation set for the small model shown in Figure 7 (although this model contains validation errors so that the UML editor would not generate any code for it). The generation set selects the package "Examples" and the three classes in that package (class A, B and C).

The "Generate" command in the UML editor brings up a window as shown in Figure 19, in which the user has to select the required generation set, defining the sub set of the model for which code has to be generated, and selects the folder (directory) in which to place the generated code file(s). This latter string is passed to the "OutputControl" context definition (see sections 3.3.2 and 6.2).

When removing the generalization from class A to class C in Figure 7, the cyclic inheritance disappears and it becomes possible to generate code for the model. It is beyond the scope of this document to describe the code generator in detail (see chapter 3 and section 6.2 for a description of the concept). The code generator in the profile used produces 4 files per class and one global file for the application. The most straightforward of these files is the one containing a constructor for each of the classes in a model. The "OutputWriter" context definition for these files is given below:
context class <<plugClass>>
outputWriter: OutputClFile(in os: TextFile,
in pAllAssociations: orderedset(Association),
in pGenClasses: Class): Boolean =

let clName = ClassNameString(),
classesSeen = superClass,
assocEnds =
(pAllAssociations
 -> select(assoc | let end1 = assoc.ownedEnd -> First(),
end2 = assoc.ownedEnd -> Last()
in (end1.type.OclAsType(Class) = self) and
(end2.isNavigable or not end1.isNavigable) and
pGenClasses -> Includes(end2.type))
-> collect(assoc | assoc.ownedEnd -> Last().type.OclAsType(Class))
-> excluding(self) - classesSeen) -> asSet()
in

os.WriteLine('{$B-}') +
os.NewLine() +
os.WriteLine('unit ' + ClassUnitNameString() + ';') +
os.NewLine() +
os.WriteLine('interface') +
os.NewLine() +
os.WriteLine('uses') +
os.WriteLine('  Classes,') +
os.WriteLine('  ObjList,') +
os.Write('  DbObject') +
(if classesSeen->NotEmpty()
then
  os.WriteLine(',') +
  os.Write('  ' + classesSeen -> First().ClassUnitNameString())
else
  0
endif)
) +
os.WriteLine(';') +
os.NewLine() +
os.WriteLine('{$I ' + FieldIDFileName() + '}') +
os.NewLine() +
os.WriteLine('type') +
(os.Write('  ' + clName + ' = class(') +
(if assocEnds -> Empty() cl
then
  os.Write('CTDbObject')
else
  os.Write(classesSeen -> First().ClassNameString()))
end)
) +
os.WriteLine('public') +
os.WriteLine('constructor Create(nObjID: Integer;') +
(os.WriteLine('  pFactory: CTBaseFactory = nil;')
override;') +
os.WriteLine('  destructor Destroy; override;') +
os.WriteLine('private') +
os.WriteLine('procedure InitAttributes;') +
os.WriteLine('{$I ' + DeclarationFileName() + '}') +
os.WriteLine('end { class ' + clName + ' };') +
os.NewLine() +
}
As can be seen from this specification, an "OutputWriter" context declaration is a combination of OCL navigation and set operations with calls to predefined operations on a "TextFile" variable. This "OutputWriter" context declaration is called from the "OutputControl" context declaration, which in turn was called through the "Generate" command in the UML editor.

This "OutputWriter" is also the most straightforward and its code generation pieces are quite easy to follow. This may be a great advantage for applications where any change in the model must be traceable into the code.

The generated code is Delphi Pascal. This gives the following result for the class B (from Figure 7, where the inheritance between class A and class C has been removed):

```delphi
{$B-}
unit ClB;

interface

uses
  Classes,
  ObjList,
  DbObject,
  ClA;

{$I DbB.fid}

type
  Cls_B = class(Cls_A)
    public
      constructor Create(nObjID: Integer;
                        pFactory: CTBaseFactory = nil): override;
      destructor Destroy; override;
    private
      procedure InitAttributes;
    end { class Cls_B };
```
implementation

uses
  Factory;

{$I DbB.imp}

constructorCls_B.Create;
begin
  inherited;
  InitAttributes
end {Cls_B.Create};

destructorCls_B.Destroy;
begin
  inherited
end {Cls_B.Destroy};

procedureCls_B.InitAttributes;
begin
  end {Cls_B.InitAttributes};
  end {unit ClB}.

It is clear that this code example is rather trivial, as the example used is very simple. However, remark that the generated code derives class B from class A (see the type declaration above), as is the case in the UML model.

As the "OutputWriter" context declaration is part of a profile and defined outside the UML editor, it is possible to define a code generator for another language than Delphi Pascal, e.g. Ada, Java or even to generate tests.

For its own projects, E2S has defined a complete Delphi Pascal code generator, which directly translates an UML model into Delphi code. The code above is an excerpt from this code generator.
9 Runtime (OCL) Constraints

9.1 Defining Runtime OCL Constraints
Classifiers, attributes and operations in a model may contain one or more runtime OCL constraints. Currently, these are restricted to invariants, that is, they have to return a Boolean value.

To enter such a constraint, double click on an object (e.g. a class) in a diagram or from the tree view and select the tab "Constraints" in the window that comes up. At the bottom of that window, the application user may define runtime constraints, by clicking on the "Add" button. This brings up a second window in which to enter the constraint text. Figure 20 summarizes the process:

Figure 20: Adding a runtime constraint

A runtime constraint may be given in OCL, as a free text or as source code. In case the user wants to enter OCL text, the object context, derived from the model, becomes important (see Figure 21):

- The selected class is the direct context for the OCL text.
- The class may contain attributes, such as the attribute "a1", derived from the class "O".
- The class may have associations with other classes. When these associations have named association ends, such as "bb" of the association "AB", these appear as relations of the object. In Figure 21, the relation "bb" is an ordered set of classes "B".
- The class may contain operations.
The OCL text may use all elements of the context shown. The OCL text in Figure 21 states that the application must contain at least one object of the class B associated with an object of the class A.

When finished entering an OCL text, the Validate button checks it for its syntactic correctness. When entering free text or source code, there is no validation.

9.2 Evaluating Runtime (OCL) Constraints

As runtime constraints address application model objects, they cannot be evaluated during the modeling process. Instead, they have to be transformed or copied (in case of free text or source code constraints) into application code. The way to do that is through OCL code in the generators:

- As explained in section 9.1, runtime constraints may be defined for classifiers, attributes and operations. So, when navigating to such an UML object, the relation "constraint" returns the list of constraints associated to that UML object. Iterating through this list, gives the single constraints:

  ```
  context Class <<PlugClass>>
  outputWriter: OutputConstraints(in os: TextFile): Integer =
  constraint->iterate(cns; acc: Integer = 0 | …)
  ```

  In the OCL text above, the iterator "cns" contains a constraint from the list of constraints associated to a given class ("context Class")

- An UML constraint has two attributes of interest for a generation:
  - "ContentKind", is in integer attribute describing the kind of runtime constraint (0 = OCL constraint, 1 = free text and 2 = source code).
  - "Body" is a string attribute containing the ASCII text of the runtime constraint. For free text and source code runtime constraint, this is sufficient to copy the text into a generation file.
• An UML constraint also has the operation "GenerateCode", which transforms an OCL constraint into a tree of OCL model elements (see further), which can be interpreted for code generation. This operation contains two parameters:
  - A text file on which to write the generated code for the runtime OCL constraint.
  - A string value defining the operation to call for the generation of OCL model elements (see further).

Figure 22 shows the OCL code to navigate through the list of runtime constraints and generate the required code.

![Image of UML constraint with GenerateCode operation]

Figure 22: Navigating through the constraints and generating code

• The call to "GenerateCode" transforms a runtime OCL constraint into an expression tree. The elements of this tree are OCL model elements, corresponding to the OCL meta-model. This model defines different kinds of OCL expression meta-elements: literals, sets, if-expression, let-expression, ...etc. All of these meta-elements are derived from the single meta-element "OclExpression", as shown in Figure 23.

• To generate code for this expression tree, the user needs to know what kind of OCL model element the top of that tree is and how to navigate through that tree:
  - As a runtime OCL constraint may be any kind of OCL expression, the top of the expression is may be any kind of OCL model element.
  - For a given OCL model element, the OCL meta-model defines how to navigate to the child and sibling elements. For example, in Figure 23, an OCL variable expression refers to a variable, which in turn may have an initialization expression.
• For each kind of OCL expression from the OCL meta-model, the profile editor may contain OCL generative code, as shown in the overview of Figure 24. The generator "PP" is called through the special operation "GenerateCode", using its second parameter to locate the required generator. This allows defining several generators for the same runtime OCL constraint (if necessary).
• The OCL code for the generator is straightforward:
  
  ```java
  context OclExpression
  outputControl: PP(in os: TextFile): Integer =
  DoPrettyPrint(os)
  ```

  As the context for this code is an OclExpression and all OCL model elements are derived from this element, this generator is called for any kind of OCL model element.

• The generator in turn calls a writer called "DoPrettyPrint". If there is a writer with that name for each OCL model element, the writer corresponding with the kind of OCL model element at the top of the expression tree will be called. Suppose the runtime OCL constraint is an if expression, the top of the expression tree will be an OCL model element called "IfExp" (see Figure 23). As there is a write for this OCL model element (see Figure 24), it is called from the generator.

• Once in a given writer, it is now possible to navigate through its child or sibling elements. Figure 25 shows the generative code for an OCL if expression:
  - The writer code illustrates how to produce a pretty printed version of the OCL if expression. For a real generator, the writer code should produce real application code.
  - An OCL if expression has three sub-expressions: the condition, the then-expression and the else-expression (see the right part of Figure 25).
  - The writer code calls the writers associated to these three sub-expressions. Note that the top of these three sub-expressions may be any kind of OclExpression. Calling their "DoPrettyPrint" automatically executes the writer corresponding with the kind of OCL model element at the top of their expression tree.
  - If there is no writer for a given kind of OCL model element, the writer from the parent is taken. If there is no writer at all, not even for OclExpression, a runtime error will occur during the execution of the generator.

![Figure 25: A writer to navigate to the child OCL model elements of an OCL if expression](image)