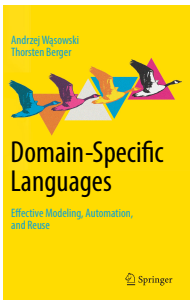


BOOK APPENDIX

Appendix D: Xtext

Andrzej Wąsowski¹ and Thorsten Berger²

Version 1, 2023-08-01



Andrzej Wąsowski and Thorsten Berger. *Domain-specific Languages: Effective Modeling, Automation, and Reuse*. Springer Nature. 2023.

Book website: <http://dsl.design>

Suppl. material: <https://bitbucket.org/dslsdesign/dslsdesign>

¹IT University of Copenhagen, Denmark

²Ruhr University Bochum, Germany

AC.1 Syntax Overview

In this chapter we show a syntax overview of Xtext with a meta model of a real-world example. Instead of using the generated Xtext code, we will rewrite the default grammar, generated by Xtext, into a simpler, more human-friendly one. A hands-on guide for using Xtext in Eclipse, with instructions on how to exactly generate the infrastructure and to run the generated editor is given in Chapter AD.

The Xtext specification language is a variation of the familiar Extended Backus-Naur Form (EBNF) notation for context free grammars. EBNF is used by most parser generators, and it is included in curriculum of most compiler courses.

We shall now discuss the main syntactic elements of the Xtext language, by presenting a simple grammar based on a tiny subset of the IUPAC nomenclature, whose meta model is shown in Fig. AC.3.

“IUPAC” is the abbreviation for “International Union of Pure and Applied Chemistry” and is a nomenclature system, that is widely used in the chemistry. Main goal of this naming system is the encoding of chemical structures in a text-based name.

The two main components are chains and branches of carbon (C) atoms. One part of the molecule is defined as chain and the rest are the branches. Every branch has the information on which carbon atom on the chain is connected to. The length information is simply substituted with a name. See table AC.1 for the substitution of chains and table AC.2.

Number of C atoms in a chain	Encoded name
1	Methan
2	Ethan
3	Propan
4	Butan
5	Pentan
6	Hexan
7	Heptan
8	Octan
9	Nonan
10	Decan

Table AC.1: Table of the first 10 chain names

For the encoded branch names the postfix “an” of the encoded chain names is replaced with “yl”.

Table AC.2: Table of the first 4 branch names

Number of C atoms in a branch	Encoded name
1	Methyl
2	Ethyl
3	Propyl
4	Butyl

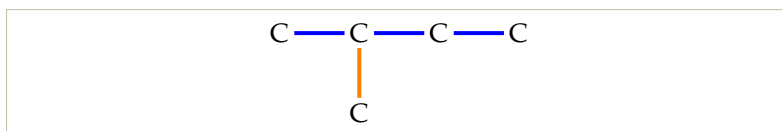
Usually if a branch with the same length occurs multiple times, these branches are summarized in one branch term and the number of branches with this length is also substituted with a string. The table AC.3 shows the first four elements. We call this element “summary prefix”.

Table AC.3: Table of the first 4 summary prefixes

Branches with the same length	Prefix
1	Mono
2	Di
3	Tri
4	Tetra

In the following example the chain is blue and the branch are drawn orange.

Figure AC.1: Chain length 4; One branch with the length 1 on position 2
Result:
2-MonoMethylButan



The order of the components are: Position(s) + minus sign + summary prefix + encoded branch length + encoded chain length. In AC.1 we have:

- Chain length 4 → Butan
- One branch with the length 1 → MonoMethyl
- Position of this branch: 2

The name assembled together: 2-MonoMethyl Butan

In the case, that there is no branch, only the encoded chain length literal will be used. So the name for AC.1 without branches is only: Butan.

The example AC.2 shows, that the order of summary prefixes and branches is also valid for multiple different branches. There are:

- Chain length 7 → Heptan

- Two branches with the length 1 → DiMethyl
- Positions of these branches: 2 + 3
- One branch with the length 2 → MonoEthyl
- Position of this branch: 4

The assembled name is: 4-MonoEthyl-2,3-DiMethyl Heptan

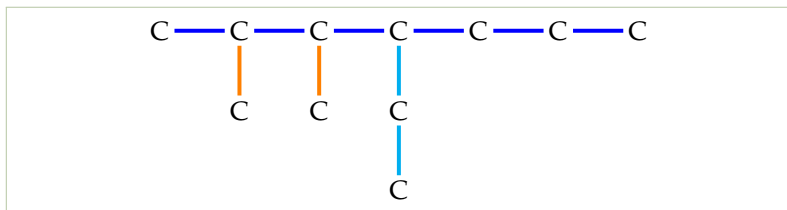


Figure AC.2: Different branch types are concatenated with a minus char. Result: **4-MonoEthyl-2,3-DiMethylHeptan**

Figure AC.3 shows a possible abstract syntax (the meta model) of the IUPAC nomenclature.

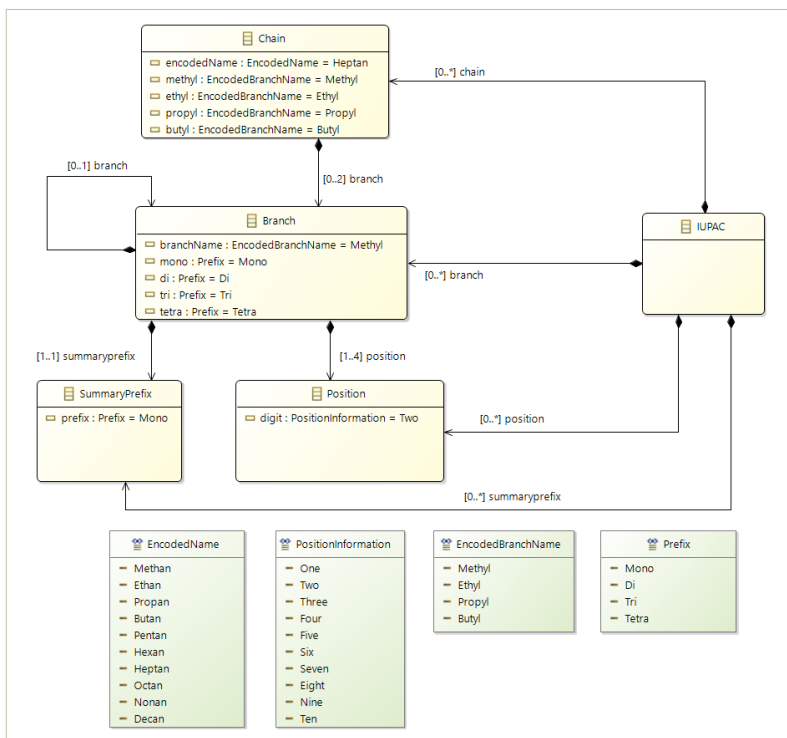


Figure AC.3: Metamodel of the IUPAC nomenclature.

Terminals are simply introduced as string literals. Enums are in this context also kind of terminals. For the enum *Prefix*, the code look like:

Figure AC.4: Code of the class SummaryPrefix

```
1 enum Prefix returns Prefix:
2   Mono = 'Mono' | Di = 'Di' | Tri = 'Tri' |
3   Tetra = 'Tetra';
```

Here is important to notice, that the element left from the equal char is one of the enum entries and on the right side is the corresponding string literal. This double usage in Xtext sounds unnecessary, but the reason for this syntax decision is, that an enum entry could also have corresponding literals, that are different from the enum entry name.

Non terminalsymbols uses the property name, that was defined in the meta model. In addition the type of the property will be included here. The non terminal “Position” could have the following implementation:

Figure AC.5: Possible implementation of Position

```
1 Position returns Position:
2   digit=EInt ("," digit+=EInt)*;
```

We see here also the grouping operator and the repetition-symbol from EBNF. The += operator adds content to a property instead of assigning. A value resulting from parsing a non terminal can be stored directly in a property of the current abstract syntax object. The meaning of the complete non terminal: *A digit plus unlimited optional digits, that needs to be started with a comma char.*

Due to the composition relations between the classes, the class “IUPAC” is the class highest order and contains all other classes with direct or indirect relations. We see in the description of the IUPAC nomenclature, that only one instance of the IUPAC class is mandatory. In Xtext such a mandatory can be expressed with braced char:

Figure AC.6: IUPAC class is mandatory to Object instantiating of EOL or Chain

```
1 IUPAC returns IUPAC:
2   {IUPAC}
3   EOL *
4   ((chain+=Chain)EOL+)*;
```

Don’t be confused with the repetition meaning in EBNF! In Xtext a repetition can only be written with the char * and + (for exact one repetition). The question mark has the same meaning compared to EBNF; it describes that an terminal or non terminal is optional. The table AC.4 shows the differences of EBNF and Xtext.

Notation	EBNF	Xtext
{...}	Repetition	Mandatory class instance
[...]	Optional	Cross-reference
,	Concatenation	n/a (invalid syntax)
=	Definition	Assignment
(*...*)	Code comment	n/a (invalid syntax)
?...?	Special sequence	n/a (invalid syntax)
-	Exception	n/a (invalid syntax)

Table AC.4: Table of EBNF notations and the meaning in Xtext

There is much more to Xtext than we present, but this is sufficient for creating simple languages. Among other elements, of the highest interest are probably customizable scoping semantics (what names are visible in what scopes), and fully qualified name support for references across name spaces/scopes. These are described in the Xtext documentation.

Finally, C-like comments (both block comments, and line comments) are automatically supported in languages built with Xtext

AC.2 Creating DSLs with Xtext

We give a brief guide to Xtext by showing how to automatically derive a concrete textual syntax for the example from the last chapter. Xtext supports both the grammar-first and the meta-model first ways of designing DSLs. In the former case, it automatically generates the meta-model from the grammar, while in the latter case, it automatically generates a crude grammar from a meta-model. We use the Ecore model of Figure AC.3 as the meta-model for this guide.

The code generation from a Ecore model to Xtext code is not done with a single operation in Eclipse. Instead of that three intermediate steps are necessary. Figure AC.7 shows the workflow:

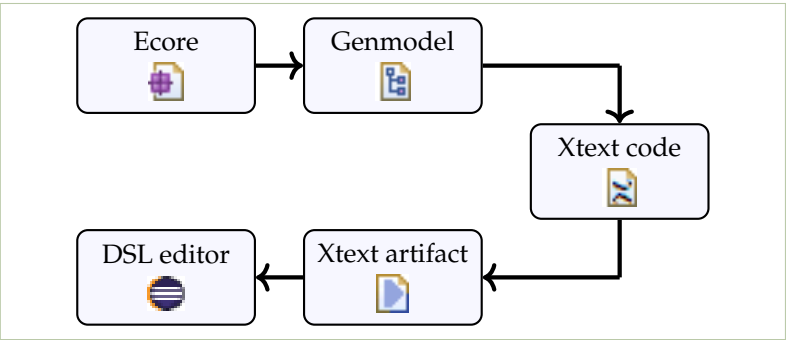


Figure AC.7: Workflow from an Ecore model to the corresponding DSL editor.

1. Model code generation

Open the context menu from the .ecore file. New → Other → EMF Generator Model. Click next in the wizard until the window with the title *Ecore Import* appears. (See figure AC.8) Check whether the right Ecore model is already given (in our case it is the file IUPAC.ecore). If this is not the case, navigate with the button *Browse Workspace ...* to the Ecore file.

Now it is necessary to use the button with the label *Load*. After that the EMF Generator Model wizard can be completed with *Next* and *Finish*.

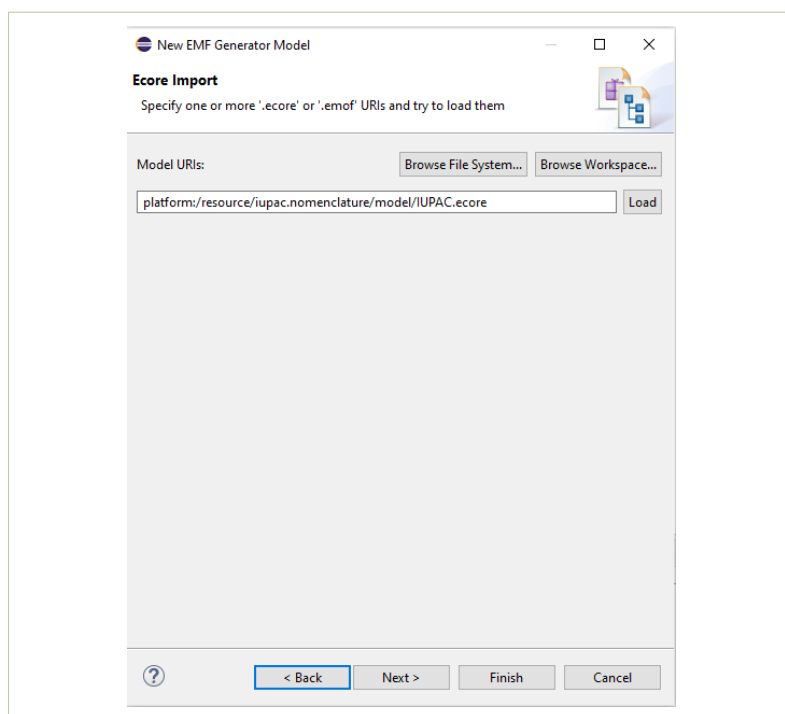


Figure AC.8: Screenshot of the Ecore Import window with the button to choose an Ecore model.

After creating the genmodel, open it and right-click on *IUPAC* in the Editor Window. Choose in the context menu *Generate Model Code* (Figure AC.9). After that you can see in the *src* folder the generated model code from the *IUPAC* package.

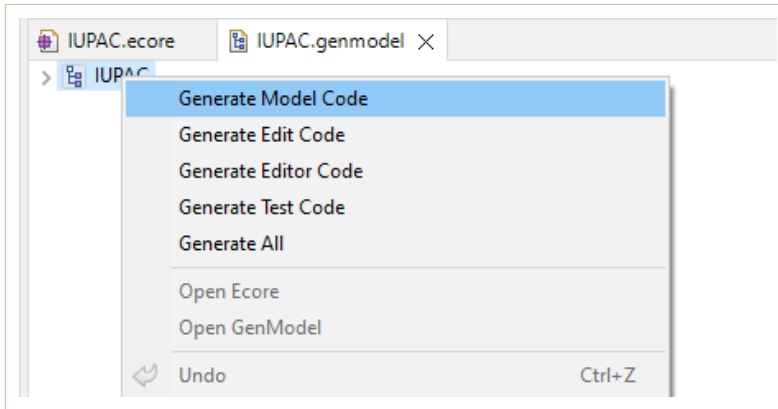


Figure AC.9: Screenshot of the Ecore Import window with the button to choose an Ecore model.

2. Create Xtext project

Now the projects from existing ecore model can be created. For the next step a new project is necessary (File → New → Project). Select the wizard *Xtext Project From Existing Ecore Models*. Add on the next page the genmodel, that was created in the previous step with the *Add* button. Now the correct entry rule needs to be selected. In our case it is the main class *IUPAC*. After selecting the genmodel and the matching entry rule the window should look like in the figure AC.10. Finish the wizard. If everything worked as expected a new project is created, and the default grammar specification for your language was generated and opened in a text editor. This generated (and clumsy) syntax is shown in figure AC.15.

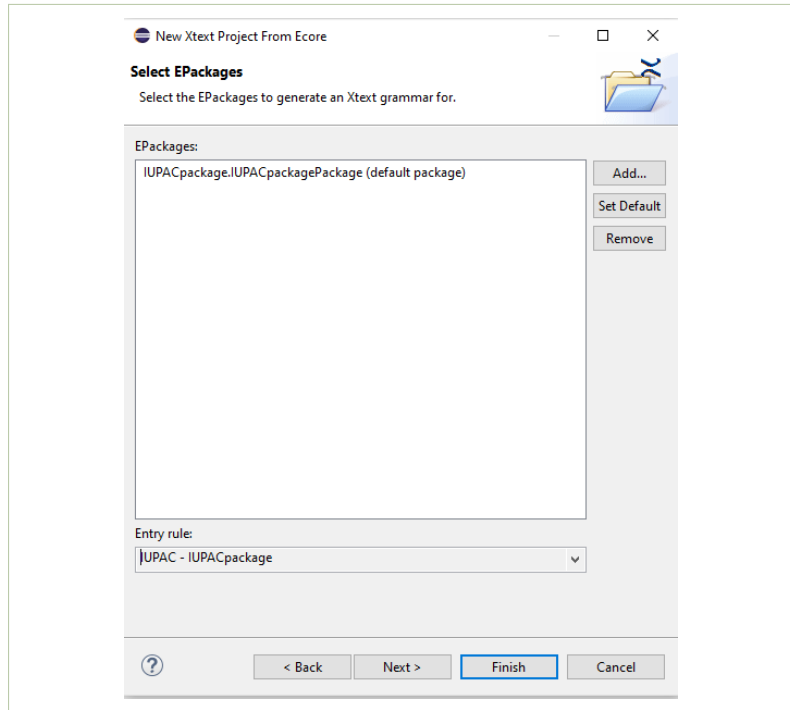


Figure AC.10: By default the first class will be selected as entry rule. In our case IUPAC needs to be selected.

3. Xtext artifacts generation

Replace the generated Xtext code with the given IUPAC Xtext code. You can find the Xtext file under the *src* folder of the *org.text.example.mydsl* project folder. Open the context menu of the Xtext file and select: Run as → Generate Xtext Artifacts.

4. Launching editor with DSL

Launch the main Xtext project as an Eclipse application (Open context menu: Run as → Eclipse application). A new instance of Eclipse should start. In the console of the first Eclipse instance could occur Java warnings. They can be ignored. In the new instance create a Java project. After that a file for our DSL is required. Create them **in the src folder**. (Via context menu: New → File) By now the project is a Java project. To automatically convert it to a suitable Xtext project, name the file with the extension **.mydsl**. After that a message box will appear (See figure AC.11). Use the Yes button for the project conversion.

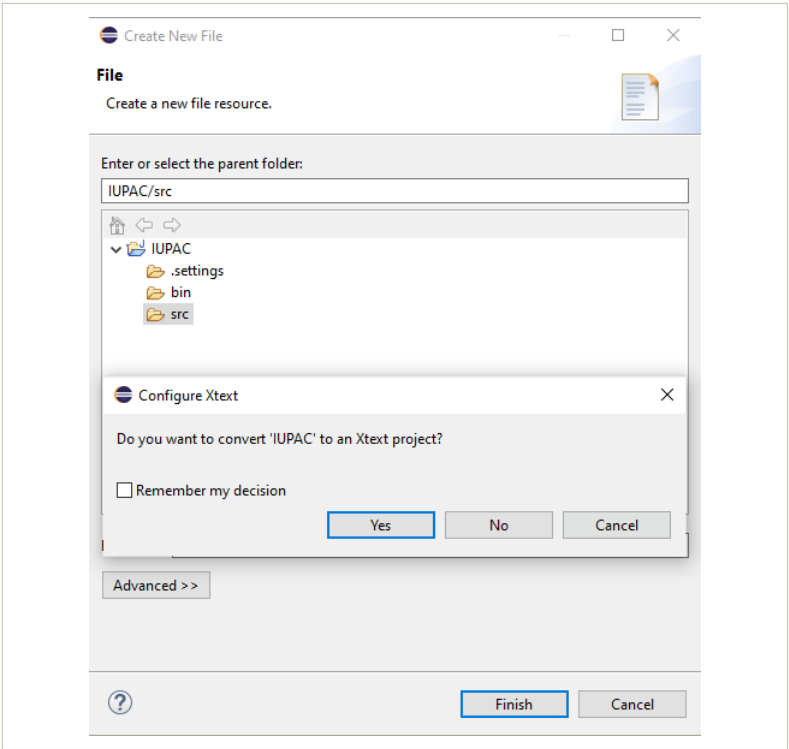


Figure AC.11: Message box, that can be only triggered with a file generation of the extension .mydsl.

Using the editor

Now the editor can be used. A simple check whether the DSL is properly implemented can be done via Ctrl+Space. This opens the syntax completion with all our enums – figure AC.12 shows this behaviour.

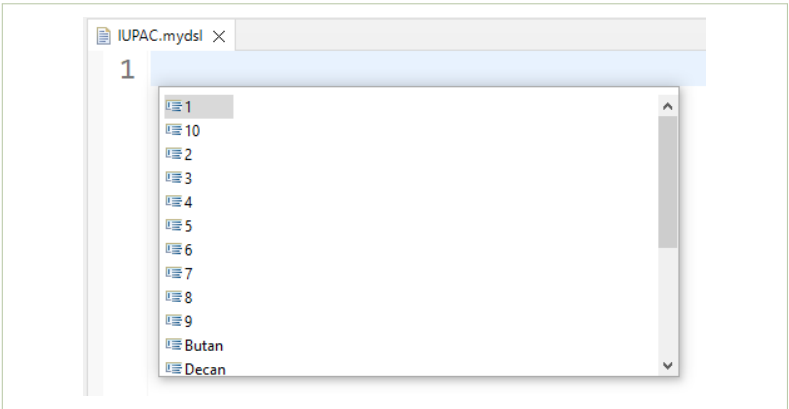


Figure AC.12: All possible enums will be shown. A similar behaviour compared to for example the java editor.

Although the created editor comes with additional functionalities (syntax highlighting and code completion), there are a few pitfalls:

- Due to historical reasons a newline is represented in different ways on different platforms. By default Xtext uses the Windows style `\r\n`. This causes to errors in Unix-like and Mac-OS systems.
- One peculiarity of Unix-like systems is, that a file is always ended with a `\n`. The editor includes automatically such a char on Unix-like systems. But with the default Windows style this is an wrong symbol.

All of the previous mentioned pitfalls can be solved with the following adjusted production rules.

Figure AC.13:
Production rule, that supports Windows and Linux endings

```
1 // \r is an optional char
2 terminal NEWLINE:
3     ("\r"? "\n");
```

Figure AC.14: Usage of an optional end of file symbol

```
1 // End of file <EOF> is an completely optional rule
2 EOF:
3     "\n"?;

5 // Main class now accepts an EOF
6 IUPAC returns IUPAC:
7     {IUPAC}
8         EOL*
9         ((chain+=Chain)EOL+)*
10        EOF;
```

Updating a DSL

Unfortunately the update process is laborious. If only the Xtext code was updated the step 3. and 4. needs to be done again. With an updated Ecore model all steps have to be repeated. Reason for this decision in Eclipse comes from the model-first approach, that usually an Ecore model will be developed first; before any Xtext code will be written.

Figure AC.15: The clumsy generated Xtext code.

```
1 // automatically generated by Xtext
2 grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals
3
4 import "http://IUPAC.ecore"
5 import "http://www.eclipse.org/emf/2002/Ecore" as ecore
6
7 IUPAC returns IUPAC:
8 {IUPAC}
9 {IUPAC}
10 '{'
11 ('chain' '{' chain+=Chain ( "," chain+=Chain)* '}' )?
12 ('branch' '{' branch+=Branch ( "," branch+=Branch)* '}' )?
13 ('position' '{' position+=Position ( "," position+=Position)* '}' )?
14 ('summaryprefix' '{' summaryprefix+=SummaryPrefix ( "," summaryprefix+=SummaryPrefix)* '}' )?
15 '}' ;
16
17 Chain returns Chain:
18 {Chain}
19 {Chain}
```

AC.3 Static semantics with OCL

OCL (*Object Constraint Language*) is for defining the static semantic rules for the DSL. To add those rules, the textual editor of Ecore can be used. There so-called OCL constraint can be implemented to define fine-granular, how the IUPAC name should look like. Open the context menu from the .ecore file. Open With → OCLinEcore Editor.

Using the OCL editor

Basically these constraints are added in a specific class. A constraint is expressed as invariant in the editor as you can see in figure AC.16. These are triggered automatically as soon as something is written to the DSL editor. The constraints presented down below are sorted by ascending complexity in code.

1. Rule: Branch names cannot be identical.

A maximum of two branches can be added to a chain. If two branch names are specified, a condition is triggered: the two branches must not have the same name. As all invariants are always checked, *true* must always be returned in the else case, if only one branch is present, as the condition can never be violated with a single branch.

You can write user-defined error messages in brackets next to the invariant name. They will be displayed in the DSL editor if the constraint is not fulfilled.

```
1 /* First Constraint */
2 invariant constraint_1('Branch names can not be
3 identical.'):
4     if self.branch.branchName -> size() = 2 then
5         self.branch.branchName -> at(1)
6         <>
7         self.branch.branchName -> at(2)
8     else
9         true
10    endif;
```

Figure AC.16: Invariant in class Chain

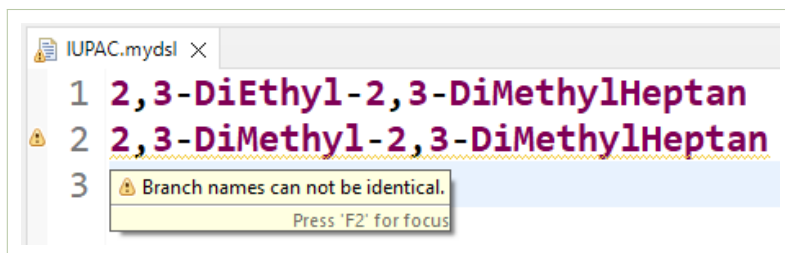


Figure AC.17: Examples in the graphical editor: The first IUPAC name satisfies the first semantic rule, while the second does not.

2. Rule: Number of position must be compatible with prefix

This rule requires attributes to be added into the *class Branch* (figure AC.18), because the value taken from the DSL Editor is of type *Prefix* and must therefore be compared with a value of type *Prefix*. Depending on the value, this is assigned to the number of positions and the following invariant (figure AC.19) is derived from this. The *and*-operator indicates that the number of positions is encoded with the specific summary prefix. With the *or*-operator we pack four conditions together, as we have up to four different prefixes, if one of them is fulfilled, the invariant returns a *true* and this constraint is then fulfilled.

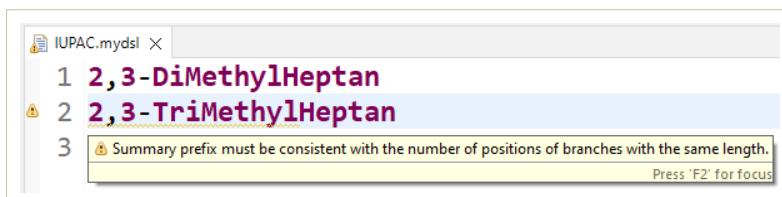
Figure AC.18: New attributes added to the class *Branch*

```
1 attribute mono : Prefix[?] = 'Mono';
2 attribute di : Prefix[?] = 'Di';
3 attribute tri : Prefix[?] = 'Tri';
4 attribute tetra : Prefix[?] = 'Tetra';
```

Figure AC.19: Invariant in class *Branch*

```
1 /* Second Constraint */
2 invariant constraint_2('Summary prefix must be consistent
3 with the number of positions of branches with the same
4 length.'):
5     (self.summaryprefix.prefix = mono
6     and
7     self.position -> size() = 1)
8 or
9     (self.summaryprefix.prefix = di
10    and
11    self.position -> size() = 2)
12 or
13    (self.summaryprefix.prefix = tri
14    and
15    self.position -> size() = 3)
16 or
17    (self.summaryprefix.prefix = tetra
18    and
19    self.position -> size() = 4);
```

Figure AC.20: Second semantic rule is not fulfilled by the IUPAC name in line 2.



3. Rule: Branch names must be sorted alphabetically.

As mentioned earlier, a chain can have up to two branch names. We compare all valid combinations with the branch names of the IUPAC name of the DSL editor, using the rule shown in Figure AC.22.

```
1 /* Newly added attributes */
2 attribute methyl : EncodedBranchName[?] = 'Methyl';
3 attribute ethyl : EncodedBranchName[?] = 'Ethyl';
4 attribute propyl : EncodedBranchName[?] = 'Propyl';
5 attribute butyl : EncodedBranchName[?] = 'Butyl';
```

Figure AC.21: New attributes added to the class Chain

```
1 invariant constraint_3('Branch names must be sorted
2 alphabetically.'):
3   if self.branch.branchName -> size() = 2 then
4     self.branch.branchName -> at(1) = butyl
5     and
6     self.branch.branchName -> at(2) = ethyl
7   or
8     self.branch.branchName -> at(1) = butyl
9     and
10    self.branch.branchName -> at(2) = methyl
11  or
12    self.branch.branchName -> at(1) = butyl
13    and
14    self.branch.branchName -> at(2) = propyl
15  or
16    self.branch.branchName -> at(1) = ethyl
17    and
18    self.branch.branchName -> at(2) = methyl
19  or
20    self.branch.branchName -> at(1) = ethyl
21    and
22    self.branch.branchName -> at(2) = propyl
23  or
24    self.branch.branchName -> at(1) = methyl
25    and
26    self.branch.branchName -> at(2) = propyl
27  else
28    true
29  endif;
```

Figure AC.22: Invariant in class Chain

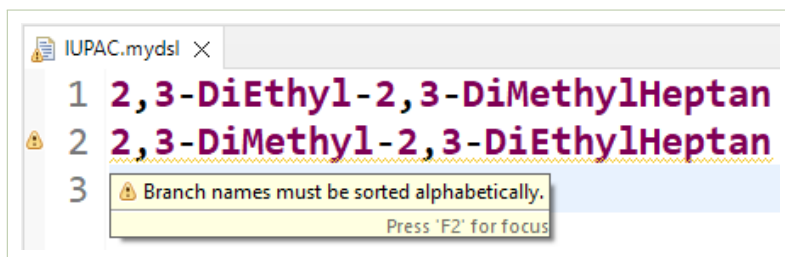


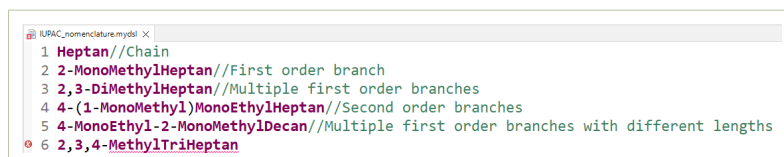
Figure AC.23: Third semantic rule is not fulfilled by the IUPAC name in line 2.

AC.4 Possible use cases of the IUPAC DSL

1. Validation of IUPAC names

One obvious use case is the validation of given IUPAC names regarding syntax and semantics. In figure AC.24 is shown, that the generated editor indicates wrong names, which are not syntactically correct. In figure AC.23, an example of a semantic error in an IUPAC name is depicted in the graphical editor.

Figure AC.24: Graphical editor highlights the negative example, where summary prefix and branch name are swapped.



```

1 Heptan//Chain
2 2-MonoMethylHeptan//First order branch
3 2,3-DiMethylHeptan//Multiple first order branches
4 4-(1-MonoMethyl)MonoEthylHeptan//Second order branches
5 4-MonoEthyl-2-MonoMethylDecan//Multiple first order branches with different lengths
6 2,3,4-MethylTriHeptan
  
```

2. Categorize IUPAC names

Another use case is an extended grammar with a grouping function. This allows to group by chain name, branch name, etc. This section focuses on grouping by chain names. First of all the meta model needs to be expanded by two additional classes as illustrated in figure AC.25. The *Catalogue* class includes any number of IUPAC names with the same chain. The *CatalogueCollection* class includes multiple catalogues and individual IUPAC names.

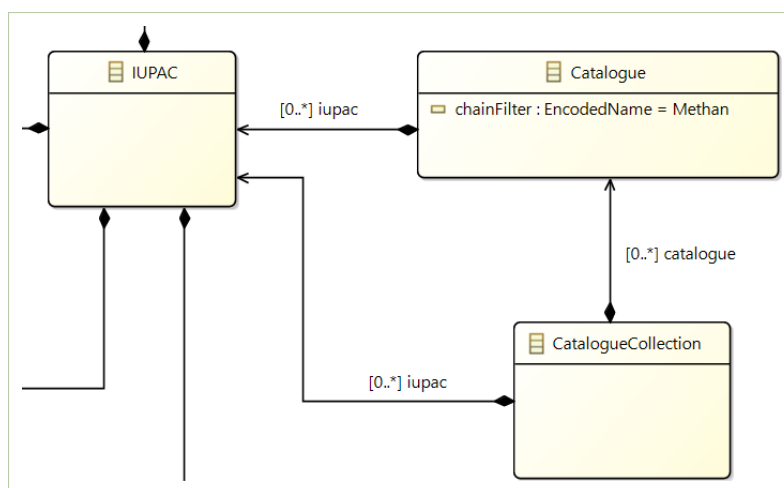


Figure AC.25: Two new classes which are linked to the IUPAC class.

After extending the metamodel, an OCL constraint (Figure AC.26) needs to be created. This constraint ensures that only chains whose encoded name matches the *chainFilter* attribute are allowed in the catalogue.

```

1 invariant FilterBy('The chain name must
2 match the chain filter.'):
3     self.iupac.chain.encodedName
4     -> forAll(name | name = self.chainFilter);

```

Figure AC.26: Invariant in class *Catalogue*

Now that the static semantics have been defined, the textual syntax needs to be adjusted via Xtext. Note that while the Xtext project is being created, the entry rule is the class *CatalogueCollection*.

Below is one of many ways of structuring the DSL syntactically so that several catalogues and several IUPAC names can be created.

```

1 IUPAC returns IUPAC:
2     ((chain+=Chain));

```

Figure AC.27: Xtext code of the class IUPAC

```

1 Catalogue returns Catalogue:
2     'Catalogue' '-' chainFilter=EncodedName
3     '\n'
4     '{'
5     '\n'
6     ((iupac+=IUPAC) '\n')*
7     '}' ;

```

Figure AC.28: Xtext code of the class *Catalogue*

```

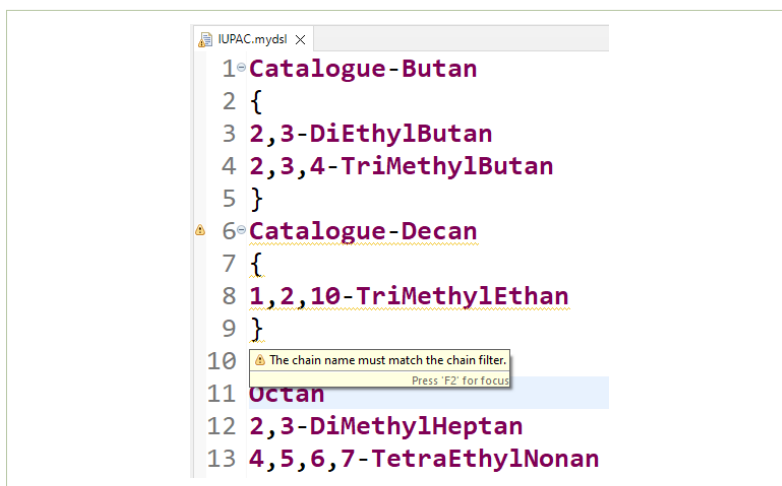
1 CatalogueCollection returns CatalogueCollection:
2     {CatalogueCollection}
3     (catalogue+=Catalogue
4     ( '\n' catalogue+=Catalogue)*)?
5     '\n'
6     (iupac+=IUPAC
7     ( '\n' iupac+=IUPAC)*)? ;

```

Figure AC.29: Xtext code of the class *CatalogueCollection*

The results in the graphical editor are depicted in Figure AC.30. A negative example is also illustrated there, where the IUPAC name does not correctly match the chain name to the filter name, which is *Decan*.

Figure AC.30:
Application of the DSL
use case in the graphical
editor, where various
catalogs are defined with
IUPAC names and
individual IUPAC names.



The screenshot shows a graphical DSL editor window titled "IUPAC.mydsl". The editor contains a list of catalog definitions, each consisting of a chain name followed by a set of individual IUPAC names in curly braces. The definitions are as follows:

- 1 Catalogue-Butan
- 2 {
- 3 2,3-DiEthylButan
- 4 2,3,4-TriMethylButan
- 5 }
- 6 Catalogue-Decan
- 7 {
- 8 1,2,10-TriMethylEthan
- 9 }
- 10 The chain name must match the chain filter.
- 11 Octan
- 12 2,3-DiMethylHeptan
- 13 4,5,6,7-TetraEthylNonan

A tooltip is visible over the "Octan" entry, displaying the message "The chain name must match the chain filter. Press 'F2' for focus."