Developer Guide BNF Tools

BNF Tools is a BNF-Editor based on xText DSLs, which gives away alot of features.

The used Features are:

- Grammardefinition
- Validation
- Quickfixing
- Generation of content from the BNF
- Formatting
- Outlining
- File import
- Deployment as Plugin
- Deployment as RCP (Rich Client Platform)

Grammardefinition:

The corefeature of xText.

This is defined in de.ugoe.cs.swe.bnftools.ebnf/de.ugoe.cs.swe.bnftools.ebnf/EBNF.xtext

IT contains the Entities after which the grammar must be defined, while the first rule is the Start e.g.:

```
EtsiBnf:
        'grammar' name=ID
        (       type='/bnf'? ';'
                (importSection=ImportSection)?
                (bnfEntry+=BnfEntry)+
        )
        |
        (       type='/delta' ';'
                (importSection=ImportSection)?
                (deltaEntry+=DeltaEntry)*
        )
        |
        (       type='/merge' ';'
                (importSection=ImportSection)?
                (mergeEntry+=MergeEntry)*
        )
;
```

To turn this into a runable application the .mwe2 file in the same folder must be executed as MWE2 Workflow.After this the whole project can be executed as an Eclipse Application for testing:

Validation allows to check for conditions in the BNF-Document:

In the File de.ugoe.cs.swe.bnftools.ebnf/de.ugoe.cs.swe.ebnf.validation/EbnfValidator.xtend

validationrules can be defined e.g.:

```
@CheckdefvoidcheckUnusedRule(Rule rule) {      var      List<RuleReference      > references = E
```

The parameter can be any Entity from the previously defined Grammar and every Entity of this Type will be checked this way.

And if the Check finds some inconsistency awarningwill be displayed to this Entity Instance in the Editor.

The other files in the Package contain supporting Methodes for the validation

like

```
EbnfAnalysisUtils.findReferences(rule);
```

or

```
EbnfAnalysisUtils.findReferences(rule,resourceDescriptions);
```

Which find the Rule references inside a BNF-File or outside a BNF-File.

Quickfixing can be applied to warnings given by Validations:

In the File de.ugoe.cs.swe.bnftools.ebnf.ui/de.ugoe.cs.swe.bnftools.ui.quickfix/EbnfQuickfixProvider.xtend

quickfixes for validation-warnings can be defined e.g.:

```
@Fix(EbnfValidator.unusedRuleDescription)
        def void fixUnusedRule(Issue issue, IssueResolutionAcceptor acceptor) {

                acceptor.accept(issue, "Remove unused rule", "Delete the unused rule", "upcase.p
                        [ element, context |
                                var Rule rule = element as Rule;
                                var IXtextDocument xtextDocument = context.getXtextDocument();
                                var ICompositeNode node = NodeModelUtils.findActualNodeFor(rule)
                                var int offset = node.textRegion.offset;
                                var String nodeText = node.text;
                                var int textLength = nodeText.length - 2;
                                xtextDocument.replace(offset, textLength, "");
                        ])
        }
```

The@Fix(*String token*)annotation definies that the following method is a quickfix for a validationwarning, with that*token*as code parameter:

```
warning(unusedRuleDescription, EbnfPackage$Literals::RULE__NAME, unusedRuleDescription, rule.nam

@Fix(EbnfValidator.unusedRuleDescription)
```

The accaptor inside applies the changes, via two possible ways:

1. Change the Document itself (like the example shows).

1. Change the underlying ecoremodel.

Generation allows to generate other files from a BNF-Document:

In our case we create a .fo document, that can be transformed into a PDF-Document using Apache FOP.

It can be customized in the File de.ugoe.cs.swe.bnftools.ebnf/de.ugoe.cs.swe.ebnf.generator/EbnfGenerator.xtend

Where thedoGeneratemethode defines how the files given by a Resource and a IfileSystemAccess should generate a new file. While for every relevant Entity from the

BNF a compile Methode handles the generation in the new file, while it calls the compile Methode for every related Entity e.g.:

```
def void doGenerate(Resource resource, IFileSystemAccess fsa,boolean mode) {
                var String workspacePath = WorkspaceResolver.getWorkspace();
                for (e : resource.allContents.toIterable.filter(EtsiBnf)) {
                        if (e.bnfEntry.size != 0) {
                                fsa.generateFile(e.name + ".fo", e.compile)
                        }
                }
        }
```

Based on the generated .fo file a PDF-document can be generated for this the class de.ugoe.cs.swe.bnftools.ebnf/de.ugoe.cs.swe.ebnf.generator/foToPDF can be used, either by giving the .fo file and the output URI without Ending or simply the giving the classpath of the file.

For this the doGenerateMethode needed an upgrade to access the filesystem via URIs:

```
def void doGenerate(Resource resource, IFileSystemAccess fsa,boolean mode) {
                var String workspacePath = WorkspaceResolver.getWorkspace();
                for (e : resource.allContents.toIterable.filter(EtsiBnf)) {
                        if (e.bnfEntry.size != 0) {
                                fsa.generateFile(e.name + ".fo", e.compile)

                                //generate pdf
                                var uri = (fsa as IFileSystemAccessExtension2).getURI(e.name + "
                                var String fullUri = workspacePath + uri.path.substring(10, uri.
                                var File file = new File(fullUri);

                                if (file.exists) {
```

```
                                        //true -> pdf, false -> rtf

                                        if(mode){
                                                FoToPdfOrRtf.createRtfFromFo(fullUri.substring(0
                                        }else{
                                                FoToPdfOrRtf.createPdfFromFo(fullUri.substring(0
                                        }

//                                      fsa.deleteFile(e.name + ".fo");
                                }
                        }
                }
        }
```

To include apache fop you need to add all the jars in a folder e.g. Libs in your project, add this folder to your buildpath, cofigure buildpath and add the jars to it and add them in the plugin.xml on the page runtime at classpath.

Formatting or Prittey Printing is to format the BNF-Document:

In the File de.ugoe.cs.swe.bnftools.ebnf/de.ugoe.cs.swe.ebnf.formatting/EbnfFormatter.xtend

the Method configureFormatting(FormattingConfig c)allows to define formatting rules

before, after or between Enteties or Keywords.

e.g.:

```
@Inject extension EbnfGrammarAccess override protected voidconfigureFormatting(FormattingConfig
{
        c.setLinewrap(0,1,2).before(SL_COMMENTRule);
        c.setLinewrap(0,1,2).before(ML_COMMENTRule);
        c.setLinewrap(0,1,1).after(ML_COMMENTRule);
        var EbnfGrammarAccess f = getGrammarAccess as EbnfGrammarAccess;c.setLinewrap.before(f.r
        c.setLinewrap.before(f.importRule);c.setNoSpace.after(f.ruleAccess.rulenumberINTTerminal
}
```

The Entities are recieved via an Inector that gives access to The Grammar.

Outlining and Labeling are Features, that show the document Structure of the BNF-Document:

Outlinining can be customized in the File

de.ugoe.cs.swe.bnftools.ebnf.ui/de.ugoe.cs.swe.bnftools.ui.outline/EbnfOutlineTreeProvider.xtend.

Here you can define a_createChildren()with the rootNode and the BNF-Entity of the Grammar to change the outline sequence:

```
def void_createChildren(DocumentRootNode parentNode, EtsiBnf bnf) {
        createNode(parentNode,bnf);
}
```

Labeling is made to customize what the outline text for an Entity should look like.

It can be customized in the file

de.ugoe.cs.swe.bnftools.ebnf.ui/de.ugoe.cs.swe.bnftools.ui.labeling/EbnfLabelProvider.xtend

Where for every Entity a text can be defined:

```
def text(ImportSection sec){'Imports'}
```

File import allows to reference Rules from one BNF-Document in another:

There are 2 ways for imports, via URI and VIA Namespaces:

The BNF-Grammar uses the URI version. To Activate this the lines

```
fragment= scoping.ImportNamespacesScopingFragmentauto-inject{}
fragment= exporting.QualifiedNamesFragmentauto-inject{}
fragment= builder.BuilderIntegrationFragmentauto-inject{}
fragment= types.[wiki:TypesGeneratorFragmentauto]-inject{}
```

in the .mwe2 file have to be commented out and the lines:

```
fragment= scoping.ImportURIScopingFragmentauto-inject{}
fragment= exporting.SimpleNamesFragmentauto-inject{}
```

must be included.

After That imports can be defined like this and will automaticly be used:

```
'import' importURI = STRING
```

Also it is possible to add features to the UI via Xtext:

Therefor i recomend reading this Guide ?http://flipsomel.wordpress.com/.

But don't use the @Override annotation!

Deployment as Plugin:

If you want to deploy your the BNF Tools you can use the deployment as plugin:

Rightclick your xTextProject, choose export, choose Plug-in development --> Deployable plug-ins and fragments, choose all parts of the project, *.ebnf *.ebnf.tests *.ebnf.ui and a directory. After you finish this will generate a jar for every one of the choosen projects. Add these to the pluigin-folder of a eclipse and it should be installed

Deployment as RCP:

If you want to create a Rich client platform for a standalone minimal worbench setup with only your plugin an requiered plugins in it RCP is a good choice (This is for an eclipse 3.x RCP).

First create your xText Project, then create a new Plug-in Project. Give it a name,

e.g. de.ugoe.cs.swe.bnftools.ebnf.product. Click next, and unchoose Generate an Activator, a Java Class that controls the plug-in-s life cycle and This plug-in will make contributions to the UI. Also choose no at Rich client Platform. Press finish.

now open the Manifest.MF, go to the Overview page and choose This plug-in in a singleton. Then go to the Dependencies page and add org.eclipse.core.runtime.

Now create a product configuration in your product project, on its Overview Pageclick new, choose a fitting name and ID, your product project as defining Plugin and org.eclipse.ui.ide.workbench as application. Now go back to the Manifest.MFand open the Extensions Page. There you should now see 1 Extension org.eclipse.core.runtime.products with a product inside. This should have org.eclipse.ui.ide.workbench as application and the given name of the product configuration as name. Rightclick the product and create a new property and if you want you can give it a customized name and value.

Now back to the p roduct configuration and its dependencies page. There you add all your xtext projects and your product, then click add Requiered Plug-ins. After this you still need to add the Plugins org.eclipse.ui.ide.application and org.eclipse.core.net. Now you can test your product by running it as a Runtime Eclipse, if there is a missing plugin you can find it using the_ _validate plugins option in the run configurations plug-ins page . Deploy it using Export as an Eclipse Product in the product configuration.

@ To make the generator run properly you need to add org.eclipse.xtext.xbase to your product configuration dependencies