Developer Guide BNF Tools

BNF Tools is a BNF editor and IDE based on xText.

Overview

This documentation includes hints related to the deployment of the tools as well as to the implementation of the following features:

- Grammar definition and code generation
- Validation
- Quick-fixing
- Generation of content from the BNF
- Formatting
- Outlining
- File import
- Deployment as Plugin
- Deployment as RCP (Rich Client Platform)

Grammar Definition and Code Generation

The xText grammer is defined in `de.ugoe.cs.swe.bnftools.ebnf/de.ugoe.cs.swe.bnftools.ebnf/EBNF.xtext`. 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

Validation allows to check for constraints and conditions within the BNF-Document. Validation rules can be defined in Xtend according to the general framework provided by xText in the file `de.ugoe.cs.swe.bnftools.ebnf/de.ugoe.cs.swe.ebnf.validation/EbnfValidator.xtend`, e.g.:

```
@Check
def void checkUnusedRule(Rule rule) {

        var List<RuleReference> references = EbnfAnalysisUtils.findReferences(rule);
        var List<Rule> references1 = EbnfAnalysisUtils.findReferences(rule,resourceDescr
        if ((references.size+references1.size == 0) && (rule.getRulenumber() != 1))
                warning(unusedRuleDescription, EbnfPackage$Literals::RULE__NAME, unusedR
}
```

The parameter can be any entity type from the previously defined grammar and the validation rule will be applied on every entity of this type. If the validation conditions are not met, a warning message will be produced and displayed next to the violating entity Instance in the editor. The other files in the package contain supporting methods for the validation, such as `EbnfAnalysisUtils.findReferences(rule)` or `EbnfAnalysisUtils.findReferences(rule,resourceDescriptions)`, which provide functionality for finding rule references within a BNF file or within a set of BNF files.

Quick-fixing

Quick-fixing can be applied to warnings raised by violations of validation constraints. Quick-fixes can be defined in Xtend according to the general framework provided by xText in the file `de.ugoe.cs.swe.bnftools.ebnf.ui/de.ugoe.cs.swe.bnftools.ui.quickfix/EbnfQuickfixProvider.xtend`, 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(EbnfValidator.unusedRuleDescription)` annotation identifies the following method as a quick-fix for the corresponding validation warning, i.e. `EbnfValidator.unusedRuleDescription` in this case, which is raised by the validation rule defined above:

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

The acceptor inside applies the changes in two possible ways:

1. Change the Document itself (as shown in the example).

1. Change the underlying Ecore model.

Generation

Generation allows the user to generate other files from a BNF document. In our case we create an intermediate `.fo` document, that can be transformed into a PDF or an RTF document by using Apache FOP. The generation of the intermediate `.fo` document can be customised in Xtend according to the general framework provided by Xtext in the file `de.ugoe.cs.swe.bnftools.ebnf/de.ugoe.cs.swe.ebnf.generator/EbnfGenerator.xtend`.

The `doGenerate` method defines how the file described by a `Resource` and an `IFileSystemAccess` shall be processed in order generate a new file in the target format. The `compile` methods then handle the transformation, for each relevant entity instance and all related entity instances, 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 with the help of the `de.ugoe.cs.swe.bnftools.ebnf/de.ugoe.cs.swe.ebnf.generator/foToPDF` class, either by providing the .fo file and the output URI without ending or simply by providing the path of the file. To enable this, the `doGenerate` method needs 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");
					}
				}
			}
		}
```

The Apache FOP libraries need to be added to the build path and the plugin dependencies.

Formatting

Formatting (or pretty-printing) can be used to automatically format the BNF document by inserting white space where appropriate in order to improve the readability of the document. Formatting can be defined in Xtend according to the general framework provided by Xtext by adapting the file `de.ugoe.cs.swe.bnftools.ebnf/de.ugoe.cs.swe.ebnf.formatting/EbnfFormatter.xtend`. The method `configureFormatting(FormattingConfig c)` describes the formatting rules before, after, or between entity instances or keywords, e.g.:

```
@Inject extension EbnfGrammarAccess
override protected void configureFormatting(FormattingConfig c)
{
        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
}
```

Outlining

Outlining and labelling are features that show the document structure of the BNF document. Outlining can be customised in Xtend according to the general framework provided by xText in the file `de.ugoe.cs.swe.bnftools.ebnf.ui/de.ugoe.cs.swe.bnftools.ui.outline/EbnfOutlineTreeProvider.xtend`. There a `createChildren()` method with `rootNode` and the BNF entity of the grammar as parameters can be defined to change the outline sequence:

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

Labelling can be customised in Xtend according to the general framework provided by xText in the file `de.ugoe.cs.swe.bnftools.ebnf.ui/de.ugoe.cs.swe.bnftools.ui.labeling/EbnfLabelProvider.xtend` in order to customise what the outline text for an entity should look like, e.g.:

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

File Imports

File imports allow referencing rules from one BNF document into another. There are two ways for imports, via URIs and via name-space. The BNF grammar uses the URI-based approach. 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 need to be commented out and the lines:

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

need to be included, after which imports can be defined in the grammar, e.g.:

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

Generation UI

See the guide at ?http://flipsomel.wordpress.com/ (skip the @Override annotations).

TODO: Transfer relevant instructions.

Deployment as Plugin

The simplest way to deploy the BNF Tools is to deploy them as a plugin:

- Right click the xTextProject --> Export... --> Plug-in Development --> Deployable plug-ins and fragments
- Choose all parts of the project, *.ebnf *.ebnf.tests *.ebnf.ui and an output location.
- Click finish.
- Transfer the generated JARs in the output location to the pluigin-folder of the target Eclipse distribution and it should be installed.

Deployment as RCP

A more portable way to deploy the BNF Tools is as a standalone Rich Client Platform (RCP) application. This results in a standalone minimal workbench setup with only the BNF plugin and required dependencies.

- Create a new Plug-in Project e.g. `de.ugoe.cs.swe.bnftools.ebnf.product`. Click next, and uncheck *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.
- Open the `MANIFEST.MF` of the newly created project, 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`.
- Create a product configuration in your product project
    - On its *Overview* page, click *New*, choose a fitting *name* and *ID*, select the product project as *defining plug-in* and `org.eclipse.ui.ide.workbench` as *application*.
- Go back to the `MANIFEST.MF` and open the *Extensions* page. There you should now see one *Extension* `org.eclipse.core.runtime.products` with a newly created product inside. This should have `org.eclipse.ui.ide.workbench` as *application* and the given name of the product configuration as *name*.
    - Right-click on the product and create a new *property* and optionally give it a customised *name* and *value*.
- Go back to the product configuration and its *Dependencies* page.
    - Add all the xText projects and the newly created product, then click *Add required plug-ins*.
    - Manually add the plug-ins `org.eclipse.ui.ide.application` and `org.eclipse.core.net`.
    - To make the generator run properly, manually add also `org.eclipse.xtext.xbase` to the dependencies.
- Test your product by running it as a *Runtime Eclipse Application*.
    - If there is a missing plug-in you can find it using the *Validate plug-ins* option in the run configurations plug-ins page.
- Deploy it using *Export as an Eclipse Product* in the product configuration.