User Guide

August 9, 2010

Contents

1 Overview

EMFText is a tool which allows to define a textual syntax for Ecore based metamodels. From a specification of this syntax, EMFText generates components to load, edit and store model instances. The syntax is specified by a so called concrete syntax specification which are stored in files with the suffix `.cs`. A `cs` specification is directly related to one or more Ecore metamodel(s) which implicitly predefine a "grammar skeleton". Figure **??** gives an overview on how the generator part of EMFText works and which components are generated.

TODO Insert figure emftext_schema.gif.

Through combining metamodel and `cs` specification, EMFText derives a context-free grammar and exports it as an Another Tool for Language Recognition (ANTLR) [**?**] parser specification. This specification contains annotated semantic actions which perform the instantiation of models. EMFText then transparently delegates parser[1] and lexer[2] generation to ANTLR by passing the generated grammar file. Since ANTLR does not cover the whole class of context-free grammars one can not guarantee the generation of a working parser for arbitrary cases. However, in most cases generation is sufficient.

While parsers are used to load model instances from textual representations a printer is needed to do the inverse (e.g., to print an in-memory model) back to a textual representation. The printed results can then again be parsed by the corresponding parser. Both instances (printed and loaded) are be equal. Furthermore, printers produce a formatted and human-readable output. The EMFText built-in printer generator achieves these goals by interpreting the `cs` file and the derived grammar. Also, `cs` specifications can be enriched by special operators to indicate that on a specific position white-spaces or newlines have to be printed. Additionally, it uses information about literals (e.g., keywords) in defined languages which are removed from model instances. As for parser generation, EMFText does not guarantee that printer generation works for arbitrary cases but mostly it is be a convenient solution.

EMFText also generates a set of resolvers. Resolvers convert parsed token strings to an adequate representation in the metamodel instance.

TokenResolvers implement a mapping from the string value of a specific token to a native Java type (e.g., boolean, int, String etc.). In the standard implementation TokenResolvers can automatically remove and add (printing) pre- and suffixes. The conversion to native java data types is done by delegation to the corresponding Java type conversion functions. For example, `Integer.parseInt("42")` results in an int valued `42`. Since this behavior is only desired for concrete syntaxes mirroring exactly (or at least partially) the Java syntax for primitive types, users are expected to implement more adequate mapping functions as needed.

Resolvements depending on context are meant to be realised by implementing ReferenceResolvers. For these only stubs are generated. While TokenResolvers are directly invoked by parsers, ReferenceResolvers are triggered on demand later by EMF's proxy resolution mech-

[1] http://en.wikipedia.org/wiki/Parser
[2] http://en.wikipedia.org/wiki/Lexical_analysis

anism. An additional feature is the evaluation of eventually annotated Object Constraint Language (OCL) [**?**] constraints. With OCL, consistency conditions can be declared on the metamodel to further improve quality of EMFText based developments.

2 Development Process

Creating parsers, printers and editors with EMFText is easy! It involves some necessary steps which are roughly depicted by the activity diagram in Fig. **??**.

TODO Insert figure emftext_process.png

2.1 Creating a metamodel

The starting point usually is the *Ecore model*. It serves as abstract syntax and as a skeleton for concrete syntax. The model can be a graph or tree definition. Although Ecore models are specified as XML files, it is recommended to use the Ecore model editor or a graphical editor (e.g., the Ecore Tools). Additionally, a unique namespace (property NS URI in the .ecore file) for the models as well as a package name needs to be specified. To enable EMFText to use models at runtime, a model plug-in must be generated. For this, EMF provides a *generation metamodel* (GenModel) allowing to enrich Ecore models with specific information for code generation. The EMF Model wizard facilitates the creation of GenModels.

The next step is to define the base package name which will be the common prefix for all subsequently generated Java packages. After finishing configuration, the *EMF model plug-in* is ready to be generated. EMFText will later use these classes to construct a model instance.

2.2 Specifying Concrete Syntax

After defining a metamodel, we can start specifying our *concrete syntax*. As a starting point, EMFText provides a syntax generator that can automatically create a concrete-syntax specification using HUTN (Human-Useable Textual Notation) from the metamodel. To manually specify the concrete syntax, a textual specification has to be written, which has the ending `cs` (from `c`oncrete `s`yntax).

It consists of four sections: In the first part, a unique name refers to the syntax as a specific resource type. Furthermore, the metamodel, whose textual syntax shall be defined, has to be referenced by its unique namespace URI and a start symbol must be selected from the model elements. In the second part, productions from other concrete syntax specifications can be imported. This is especially useful if the metamodel is compositionally structured, e.g. reuses large parts from other models which have an already specified concrete syntax. The third part allows for specifying token types as it is usual for parser generators, but in contrast, EMFText also allows to leave them out: If no token definitions are given, default rules are used. Another specialty for token definitions are optional pre- and suffixes which are transparently removed (after parsing) and added (before printing) by the generated token resolvers. And last but not least, EBNF-like productions have to be defined for each non-abstract model element reachable from the start symbol. They have to be defined with respect to the attributes and references of the model element and thus do not contain nonterminals in the classic sense.

2.3 Generate Resource Plug-ins

The context menu on concrete syntax specification files offers an item to generate the actual *resource plug-in* which contains the parser, printer and the editor for the language. In an optional last step, the generated token resolvers and the printer can be tailored to specific needs.

2.4 Customize DSL

TODO Add content

3 Concrete Syntax Specification Language (CS)

An EMFText syntax specification must be contained in a file with the extension `.cs` and consists of four main blocks:

1. A mandatory configuration block, which contains the name, the metamodel and the root metaclass (start symbol). Optionally other syntaxes and metamodels can be imported and generation options can be specified.
2. An (optional) `TOKEN` section. Here, tokens for the lexical analyser can be specified.
3. An (optional) `TOKENSTYLE` section. Here, the default style (color, font style) for tokens and keywords can be specified.
4. A `RULES` section, which defines the syntax for concrete metaclasses.

In the following sections the four main blocks will be explained in more detail.

3.1 Configuration Block

3.1.1 Required General Information

The first required piece of information, is the file extension that shall be used for the files will contain your model:

```
SYNTAXDEF yourFileExtension
```

Second, EMFText needs to know the EMF generator model (`.genmodel`) that contains the meta classes for which the syntax will be used. EMFText does use the generator model rather than the Ecore model, because it requires information about the code generated from the Ecore model (e.g., the fully qualified names of the metaclasses).

The genmodel is found using its URI. The second parameter (`yourGenmodelLocation`) is optional. It is only required, if the generator model is not registered in the generator model registry, or if the name of the syntax definition (without the file extension) is different from the generator model file name. In both cases a relative or absolute URI can be given to point to the generator model.

```
FOR <yourGenModelURI> <yourGenmodelLocation>
```

Third, the root element (start symbol) must be given. The root element must be a metaclass from the metamodel:

```
START yourRootMetaClassName
```

A CS specification can also have multiple root elements, which must be separated by a comma.

Altogether a usual header for a `.cs` file looks something like:

```
SYNTAXDEF yourFileExtension
FOR <yourGenModelURI> <yourGenmodelLocations>
START yourRootMetaClassName
```

3.1.2 Importing other Models and Syntaxes

Models and syntaxes can be imported in a dedicated import section, which must follow after the start symbols:

```
IMPORTS {
    // imports go here
}
```

The list of imports must contain at least one entry. If no imports are needed the whole section can be left out. An import entry consists of a prefix, which can be used to refer to imported elements in rules, the model URI and optionally the name of a concrete syntax defined for that model. If a syntax is imported, all its rules are reused and need not to be specified in the current CS specification. Importing the syntax is optional. One can also just import the metamodel contained in the generator model.

```
prefix : <genModelURI> <locationOfTheGenmodel>
    WITH SYNTAX syntaxURI <locationOfTheSyntax>;
```

The two locations are again optional. For resolving the generator model the same rule as for the "main" generator model apply. For locating the syntax, EMFText looks up the registry of registered syntax specifications. If no registered syntax is found, the `locationOfTheSyntax` is used to find the `.cs` file to import.

3.1.3 Code Generation Options

EMFText's code generation can be configured using various options. These are specified in a dedicated optional `OPTIONS` section:

```
OPTIONS {
    // options go here in the following form:
    optionName = "optionValue";
}
```

The list of valid options can be found in the Javadoc of the enumeration OptionTypes[1]. Search for `<!- begin-model-doc ->`.

3.2 Tokens

EMFText allows to specify custom tokens. Each token type is defined by a regular expression. This expression is used to convert characters from the DSL files to from groups (i.e., tokens).

[1] http://svn-st.inf.tu-dresden.de/svn/reuseware/tags/current_release/EMFText/org.emftext.sdk.concretesyntax/src-gen/org/emftext/sdk/concretesyntax/OptionTypes.java

Tokens form the smallest unit processed by the generated parser. By default, EMFText implicitly uses a set of predefined standard tokens, namingly:

- TEXT : ('A'..'Z'|'a'..'z'|'0'..'9'|'_'|'-')+,
- LINEBREAK : ('
  r
  n'|'
  r'|'
  n'),
- WHITESPACE : (' '|'
  t'|'
  f').

The predefined tokens can be removed using the `usePredefinedTokens` option:

```
OPTIONS {
    usePredefinedTokens = "false";
}
```

3.2.1 Defining Custom Tokens

To define custom tokens, a TOKEN section must be added to the `.cs` file. This section has the following form:

```
TOKENS {
    // token definition go here in the form:
    DEFINE YOUR_TOKEN_NAME $yourRegularExpression$;
}
```

Every token name has to start with a capital letter. A regular expression must conform to the ANTLRv3 syntax for regular expressions (without semantic annotations). However, don't worry: EMFText will complain if there is a problem with your regular expressions.

3.2.2 Composed Tokens

Sometimes, the regular expressions are quite repetitive and one wants to reuse simple expressions to compose them to more complex ones. To do so, one can refer to other token definition by their name. For example:

```
// simple token
DEFINE CHAR $('a'..'z'|'A'..'Z')$;
// simple token
DEFINE DIGIT $('0'..'9')$;
// composed token
DEFINE IDENTIFIER CHAR + $($ + CHAR + $|$ + DIGIT + $)*$;
```

If tokens definitions are merely used as "helper" tokens, they can be tagged as `FRAGMENT`. This means the helper token itself is used in other token definition, but not anywhere else in the syntax specification:

```
// simple token
DEFINE CHAR $('a'..'z'|'A'..'Z')$;
// helper token - not used on its own
DEFINE FRAGMENT DIGIT $('0'..'9')$;
// composed token
DEFINE IDENTIFIER CHAR + $($ + CHAR + $|$ + DIGIT + $)*$;
```

3.2.3 Tokens Priorities

EMFText does automatically sort token definition. However, sometimes token definitions might be ambiguous, i.e. the regular expressions defined for two different token are not disjoint. In such cases EMFText will always prefer the token defined first in the specification. By default, the predefined tokens (`TEXT`, `WHITESPACE` and `LINEBREAK`) have lower precedence than any defined token. However, they can be given a higher priority by prioritizinh them over other tokens using the following directive:

```
PRIORITIZE nameOfPredefinedToken;
```

3.3 Token Styles

To define the default syntax highlighting for a language a special section `TOKENSTYLES` can be used. For each token or keyword the color and style (`BOLD`, `ITALIC`, `STRIKETHROUGH`, `UNDERLINE`) can be defined as follows:

```
TOKENSTYLES {
    // show YOUR_TOKEN in black
    "YOUR_TOKEN" COLOR #000000;
    // show keyword 'public' in red and bold font face
    "public" COLOR #FF0000, BOLD;
}
```

3.4 Syntax Rules

For each concrete meta class you can define a syntax rule. The rule specifies what the text that represents instances of the class looks like. Rule have two sides—a left and right-hand side. The left side denotes the name of the meta class, while the right-hand side defines the syntax elements.

3.4.1 Simple Syntax

The most basic form of a syntax rule is:

```
YourMetaClass ::= "someKeyword" ;
```

This rule states that whenever the text `someKeyword` is found, an instance of `YourMetaClass` must be created. Besides text elements that are expected "as is", parts of the syntax can be optional or repeating. For example the syntax rule:

```
YourMetaClassWithOptionalSyntax ::= ("#")? "someKeyword" ;
```

states the instances of `YourMetaClassWithOptionalSyntax` can be represented both by `#someKeyword` and `someKeyword`. Similar behavior can be definied using a star instead of a question mark. The syntax enclosed in the paranthesis can then be repeated.

3.4.2 Syntax for Attributes

If metaclasses have attributes we can also specify syntax for the value of these attributes. To do so simple add brackets after the name of the attribute:

```
YourMetaClassWithAttribute ::= myAttribute[] ;
```

Optionally one can specify the name of a token inside the brackets. For example:

```
YourMetaClassWithAttribute ::= myAttribute[MY_TOKEN] ;
```

If the token name is omitted, as in the first example, EMFText uses the token predefined token `TEXT`, which includes alphanumeric characters. The found text is automatically converted to the type of the attribute. If this conversion is not successfull an error is raised when opening a file containing wrong syntax.

Another possibility to specify the token definition that shall be used to match the text for the attribute value is do it inline. For example

```
YourMetaClassWithAttribute ::= myAttribute['(',')'] ;
```

can be used to express that the text for the attribute value must be enclosed in round brackets. Between the brackets arbitrary characters (except the closing bracket) are allowed. Other characters can be used as prefix and suffix here as well.

By default the suffix character (in the example above this was the closing bracket) can not be part of the text for the attribute value. To allow this an escape character needs to be supplied:

```
YourMetaClassWithAttribute ::= myAttribute['(',')','\'] ;
```

Here the backslash can be used inside the brackets to escape the closing bracket. It must then also be used to escape itself. For example, `(some text (some more
))` yields the attribute value `some text (somemore)`.

3.4.3 Syntax for References

Metaclasses can have references and consequently there is a way to specify syntax for these. EMF distinguishes between *containment* and *non-containment* references. In an EMF model, the elements that are referenced with the former type are contained in the parent elements. EMFText thus expects the syntax of the referenced elements (children) to be also contained in the parent syntax. The latter (non-containment) references are referenced only and are contained in another (parent) element. Analogous to the containment references, EMFText

expects the actual syntax for those elements to be contained in the parent. The referenced elements are represented by some symbolic name only.

Syntax for Containment References

A basic example for defining a rule for a meta class that has a containment reference looks like this:

```
YourContainerMetaClass ::= "CONTAINER" yourContainmentReference ;
```

It allows to represent instances of `YourContainerMetaClass` using the keyword `CONTAINER` followed by one instance of the type that `yourContainmentReference` points to. If multiple children need to be contained the following rule can be used:

```
YourContainerMetaClass ::= "CONTAINER" yourContainmentReference* ;
```

In addition, each containment reference can be restricted to allow only certain types, for example:

```
YourContainerMetaClass ::= "CONTAINER" yourContainmentReference : YourSubClass ;
```

does allow only instances of `YourSubClass` after the keyword `CONTAINER` even though `yourContainmentRefer` may have a more general type.

Syntax for Non-Containment References

A basic example for defining a rule for a metaclass that has a non-containment reference looks like this:

```
YourPointerMetaClass ::= "POINTER" yourNonContainmentReference[] ;
```

The rule is very similar to the one for containment references, but used the additional brackets after the name of the reference. The brackets defined the token that the symbolic name must match. In the case above, the default token `TEXT` is used. So the syntax for an example instance of class `YourPointerMetaClass` can be `POINTER a`.

Since `a` is just a symbolic name that must be resolved to an actual model element, EMFText generates a Java class that implements the interface IReferenceResolver. This class be customized to specify how symbolic names are resolved to model elements. The default implementation of the resolver looks for all model elements that have the correct type (the type of `yourNonContainmentReference`) and that have a name or id attribute that matches the symbolic name.

3.4.4 Syntax for Printing Instructions

To print models to text, some additional information is required by EMFText. This information includes the numher of whitespaces and line breaks to be inserted between keywords, attribute values, references and contained elements. If you do not want to print models to text, printing instructions are not needed in your `.cs` file.

Syntax for Printing Whitespace

To explicitly print whitespace characters, the # Operator can be used in the right side of syntax rules:

```
YourMetaclass ::= "keyword" #2 attribute[];
```

It is followed by a number that determines the number of whitespaces to be printed. In the example above, two whitespace characters are printed between the keyword and the attribute value.

Syntax for Printing Line Breaks

To explicitly print line breaks, the ! Operator can be used in the right side of syntax rules:

```
YourMetaclass ::= "keyword" !0 attribute[];
```

It is followed by a number that determines the number of tab characters that shall be printed after the line break. In the example above, a single line break is printed after `keyword`. The number of tabs refers to the current model element (i.e., `EObject`), which is printed.

3.4.5 Syntax for expressions

To define syntax for metaclasses that represent expressions (e.g., binary, unary expressions like additive or multiplicative expressions) one can use the `Operator` annotation. This annotation can be added to all rules, which refer to expression metaclasses. For example, the rule:

```
@Operator(type="binary_left_associative", weight="1", superclass="Expression")
Additive ::= left "+" right;
```

can be used to define syntax for a metaclass `Additive`. The references `left` and `right` must be containment references and have type `Expression`, which is the abstract supertype for all metaclasses of the expression metamodel.

The `type` attribute specifies the kind of expression at hand, which can be binary (either `left_associative` or `right_associative`), `unary` or `primitive`.

The `weight` attribute specified the priority of on expression type over another. For example, if a second rule:

```
@Operator(type="binary_left_associative", weight="2", superclass="Expression")
Multiplicative ::= left "*" right;
```

is present, EMFText will create an expression tree, where `Multiplicative` nodes are created last (i.e., multiplicative expressions take precedence over additive expressions).

Unary expressions can be defined as follows:

```
@Operator(type="unary_prefix", weight="4", superclass="Expression")
Negation ::= "-" body;
```

There is also the option to define `unary_postfix` rules.

Primitive expressions can be defined as follows:

```
@Operator(type="primitive", weight="5", superclass="Expression")
IntegerLiteralExp ::= intValue[INTEGER_LITERAL];
```

They should be used for literals (e.g., numbers, constants or variables).

For examples how to use the `Operator` rules see the SimpleMath language in the EMFText Syntax Zoo[2] and the ThreeValuedLogic DSL[3]. These do also come with an interpreter which shows how expression trees can be evaluated.

3.5 Suppressing warnings

To suppress warnings issues by EMFText in `.cs` files one can use the `@SuppressWarnings` annotation. This annotation can be added to rules, token definitions or complete syntax definitions. One can either suppress all warnings or just specific types. A list of all types can be found

here. Please use lower camel-case syntax to specify the kind of option to suppress. For example, to suppress warnings about features without syntax, you may use:

```
@SuppressWarnings(featureWithoutSyntax)
MyMetaClass ::= "someKeyword";
```

[2] TODO
[3] TODO

List of Figures

List of Listings

Bibliography