ITEA Office
High Tech Campus 69 - 3
5656 AG Eindhoven
The Netherlands

T + 31 88 003 6136
E info@itea3.org
W www.itea3.org

ITEA 3 is a EUREKA strategic ICT cluster programme

D3.4 Eclipse-based Blended Modeling Generation Environment Documentation

BUMBLE

Blended Modelling for Enhanced Software and Systems Engineering

Contributors

Jörg Holtmann	University of Gothenburg
Jan-Philipp Steghöfer	University of Gothenburg
Weixing Zhang	University of Gothenburg

Reviewers

Roelof Hamberg	Canon Production Printing

Project Acronyms

<ACR>	<Acronyms>
BUMBLE	Blended Modelling for Enhanced Software and Systems Engineering
MDE	Model-Driven Engineering
DSML	Domain-Specific Modeling Language
UML	Unified Modeling Language
EMF	Eclipse Modeling Framework
UML-RT	UML for Real-time
rtUML	Real-time UML
SysML	Systems Modeling Language
xtUML	Executable UML
CNL	Controlled Natural Language
ALF	Action Language for Foundational UML
XML	eXtensible Markup Language
GMF	Graphical Modeling Framework
SoC	System-On-Chip
NoC	Network-On-Chip
ECU	Engine Control Unit
MPS	Meta Programming System
PapyrusIC	Papyrus Industry Consortium
PapyrusRT	Papyrus for Real-time
OEM	Original Equipment Manufacturer
AST	Abstract Syntax Tree

Table of contents

1. Introduction

This deliverable describes one BUMBLE solution for the generation of editors in the Eclipse technology space. Regarding the five BUMBLE features as introduced in the deliverable D2.2, the generation of editors is particularly motivated by the BUMBLE feature "Evolution (E)": Metamodels are typically subject of evolution, and evolving of hand-crafted editors based on these metamodels results in manual effort. Thus, an automatic approach for a metamodel-based editor realization like the generation would strongly reduce such an effort on the evolution of the editors due to metamodel changes.

From the BUMBLE use cases residing in the Eclipse technology space (cf. deliverable D2.2), there is currently only Use Case 3 (Vehicular Architectural Modeling in EAST-ADL) that applies an editor generation approach. Thus, we describe this approach in this deliverable in Section 2. We provide the corresponding tooling as Open Source Software[1].

2. Automatic Post-processing of a Textual Grammar based on the EAST-ADL Metamodel

2.1. Introduction to UC3: Vehicular Architectural Modeling in EAST-ADL

The *Electronics Architecture and Software Technology - Architecture Description Language (EAST-ADL)*[2] is a DSML for the specification of automotive embedded systems, which is applied at AB Volvo. As indicated in Figure 1 by the grey existing plugins, the Eclipse-RCP-based tool suite EATOP[3,4] provides tree and form editors to enable the tool-based specification of EAST-ADL models based on the Eclipse Modelling Framework (EMF)[5]. In the BUMBLE Use Case 3 and w.r.t. BUMBLE feature (B) (cf. Deliverable D2.2), we want to complement these existing editors with a textual notation and a seamless switching and synchronization between the textual representation and the tree-/form-based editors, that is, blended EAST-ADL modeling. In Deliverable D3.1, we introduce the textual editor for the textual notation *EATXT* and describe its architecture. Furthermore, Figure 1 depicts and indicates the architecture excerpt that is relevant for this deliverable.

[1] https://github.com/blended-modeling/east-adl-simplified/
[2] http://www.east-adl.info/
[3] https://www.eclipse.org/eatop/
[4] https://bitbucket.org/east-adl/east-adl/
[5] https://www.eclipse.org/modeling/emf/

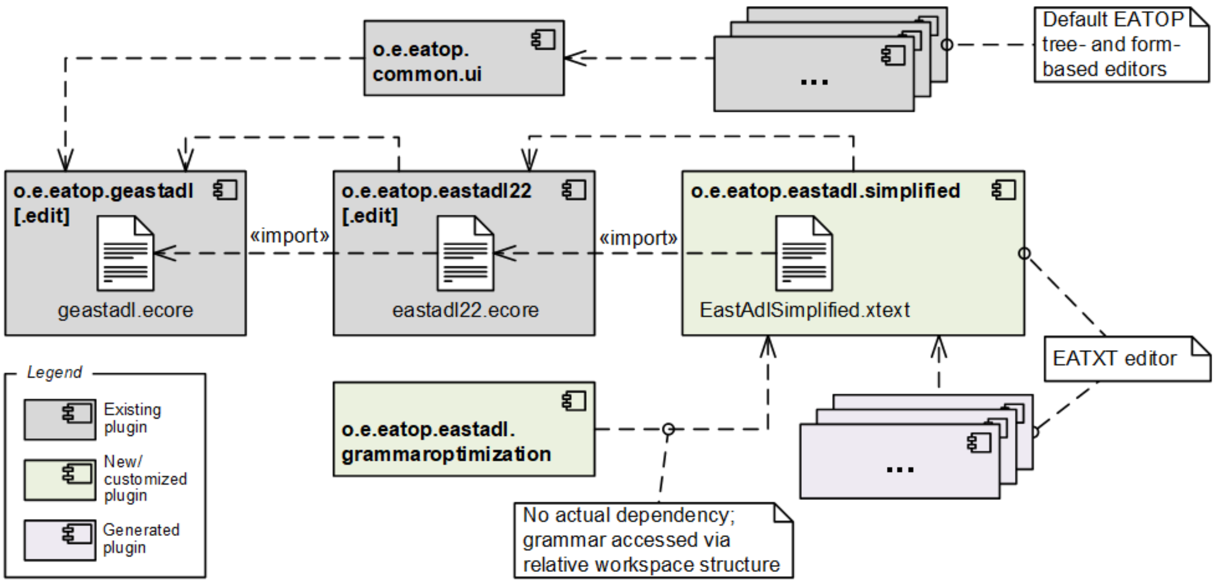
o.e.eatop.common.ui
...
Default EATOP tree- and form-based editors
o.e.eatop.geastadl [.edit]
geastadl.ecore
«import»
o.e.eatop.eastadl22 [.edit]
eastadl22.ecore
«import»
o.e.eatop.eastadl.simplified
EastAdlSimplified.xtext
EATXT editor
Legend
Existing plugin
New/ customized plugin
Generated plugin
o.e.eatop.eastadl.grammaroptimization
No actual dependency; grammar accessed via relative workspace structure
...

Figure 1: Architectural relationships of the plugin that post-processes the grammar of the EATXT editor based on EATOP

We apply the language engineering framework Xtext[6] for the realization of EATXT, which is depicted in Figure 1 as `o.e.eatop.eastadl.simplified` and is indicated by further plugins. Xtext enables the automatic generation of a grammar as well as further Eclipse plugins (which together form the textual editor) from EMF metamodels, i.e., the EAST-ADL metamodel `eastadl22.ecore` in our case (as part of `o.e.eatop.eastadl22` in Figure 1). If the EAST-ADL metamodel evolves, the EATXT grammar that depends on it has to be adapted to this evolution. The EATXT language design as coordinated with AB Volvo requires many adaptations to the grammar that is initially generated by the Xtext framework from the EAST-ADL metamodel. To efficiently incorporate these adaptations into the grammar in case the EAST-ADL metamodel evolves, we developed a post-processing plugin that automates these grammar adaptations.

In the remainder of this deliverable and complementing the architectural descriptions in D3.1, we describe the functional principles of the automatic post-processing of EATXT's grammar. Figure 1 depicts the corresponding plugin as `o.e.eatop.eastadl.grammaroptimization`.

2.2. Post-processing

Xtext generates a default grammar (which is usually in an *.xtext file), but the default grammar is complex, not user-friendly and not easy to use. In particular, it contains many unnecessary keywords, structures the input using brackets, and enforces a very verbose style of writing. The default grammar therefore needs to be optimised to make it easier to use, user-friendly and in line with the needs expressed by AB Volvo in BUMBLE Use Case 3. To this end, we first designed the grammar style, and then created a processor to transform the generated grammar into the style we want.

2.2.1. Rationale of Design Decision

To get the grammar we want, there are two technical possibilities (see the Figure 2):

[6] https://www.eclipse.org/Xtext/

1. Change the grammar generator of xtext framework, to make it directly generate grammars in the style we want.
2. Change the content of the *.xtext file after it is generated.

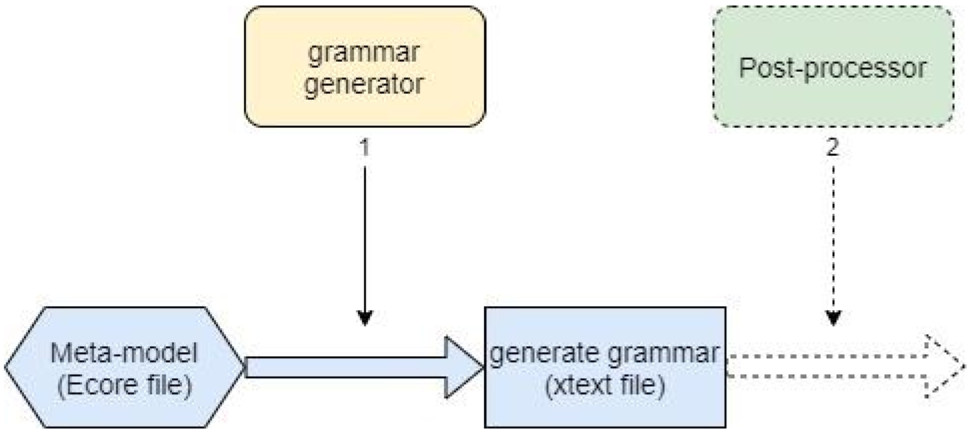

Figure 2: Two alternative realization options for the adaptation/evolution of a metamodel-based grammar in the context of Xtext

The first possibility has the disadvantage that the grammar generator is dependent on the Xtext version used since grammar generators use low-level implementation details of Xtext. In addition, it is very dependent on the underlying meta-model since it provides translation rules based on the concepts in it. That means that the grammar generator has to be changed whenever either Xtext or the underlying meta-model is updated.

The second possibility is independent of the version of Xtext since it works on the generated grammar file. It thus does not use any of the implementation details of Xtext. It is also more independent of the underlying meta-model since it works on the structure of the grammar, rather than on meta-model concepts. To make this work automatically, we designed an independent plug-in, i.e., a post-processor that takes the generated grammar file and transforms the style of the grammar.

2.2.2. Design of the Post-Processor

We develop the post-processor plug-in to modify the text in the *.xtext file. The post-processing steps are as follows:

1. Remove brackets "{" and "}", and instead use indents and whitespaces to implement code hierarchy, akin to Python.
2. Remove commas ",", and instead use newlines to distinguish different entries on the same level.
3. Allow data structures which contain no elements when all of their elements are optional.
4. Remove unnecessary keywords, such as `shortName`, `subPackage`, `topLevelPackage`, `EAXML`, `port`, `connector`, etc., which makes it less nested.
5. Implement a rule for data types such as `Numerical` which need to be completed or implemented after the grammar is generated.

Only steps 4 and 5 are dependent on the meta-model, while the rest is meta-model agnostic. Step 1 is important to realize a Python-like style. The principle idea is the application of whitespace-aware parsing. The steps to apply this technique are as follows:

1. Use synthetic tokens in the grammar by writing terminals (used to mark the boundaries of code blocks) as follows:
    - `terminal BEGIN: 'synthetic: BEGIN';`
    - `terminal END: 'synthetic: END'.`
2. Inherit expressions from Xbase and redefine the syntax of block expressions as follows:

```
@Override
XBlockExpression returns xbase::XExpression:
        {xbase::XBlockExpression}
        BEGIN
                (expression+=XExpressionOrVarDeclaration ';' ? )*
        END;
```

3. At the beginning of the *xtext file (i.e. grammar definition file), import Xbase, i.e. http://www.eclipse.org/xtext/xbase/Xbase, which is a statically typed expression language for Java that is implemented in Xtext and can be reused (imported) in other Xtext-languages. Xbase helps us to add behaviors to our DSL such as whitespace-aware features.

2.2.3. Use of the Post-Processor

The post-processor is an independent Eclipse plugin implemented in Java. Before using it, you should make sure it has been imported into Eclipse, and its code is in the workspace where the project `org.bumble.eastadl.simplified` is located. Then the steps are:

1. Right click on the plugin project (i.e., `org.bumble.eastadl.grammaroptimization`).
2. Click "run as" and then click "Java Application"

There will be lines of log in console once you run the plugin, which informs you about the post-processing status.

Optionally, if you want to start from a freshly generated grammar:

1. Import the project `org.eclipse.eatop.eastadl<desiredVersionNumber>.`
2. Start the grammar generation via `File -> new -> Other -> Xtext Project From Existing Ecore Models:`
    a. Add `eastadl<desiredVersionNumber>.genmodel` and select `Identifiable` as entry rule.
    b. Specify `org.bumble.eastadl.simplified` as project name, `org.bumble.eastadl.simplified.EastAdlSimplifiedText` as language name, and `eatxt` as extensions.