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

BUMBLE Deliverable D3.4

Eclipse-based Blended Modeling Generation Environment Documentation

Contributors

Jörg Holtmann	University of Gothenburg
Jan-Philipp Steghöfer	University of Gothenburg
Weixing Zhang	University of Gothenburg
Gloria Ninsiima	Mälardalen University
Joakim Korhonen	Mälardalen University
Malvina Latifaj	Mälardalen University

Reviewers

Roelof Hamberg	Canon Production Printing
Federico Ciccozzi	Mälardalen University

Project Acronyms

<ACR>	<Acronyms>
BUMBLE	Blended Modeling for Enhanced Software and Systems Engineering
MDE	Model-Driven Engineering
DSML	Domain-Specific Modeling Language
EMF	Eclipse Modeling Framework
MM	Metamodel
EAST-ADL	Electronics Architecture and Software Technology – Architecture Description Language
EATOP	EAST-ADL Tooling Platform
RCP	Rich Client Platform
EATXT	EAST-ADL Textual Language
IDE	Integrated Development Environment
M2M	Model-to-Model Transformation
M2T	Model-to-Text Transformation
VSM	Viewpoint Specification Model

Versions

Release	Date	Reason of change	Status	Distribution
V1.0	30/11/2021	Initial version describing the editor generation aspects of UC3	Final	Uploaded to ITEA portal
V2.0	30/09/2022	- Document restructuring - Added editor generation aspects for UC1 (Section 3) - Added and updated editor generation aspects for UC3 (now Section 2)	Final	Uploaded to ITEA portal
V3.0	03/03/2023	Only minor changes in comparison to V2.0	Final	Uploaded to ITEA portal

Executive Summary

Handcrafting and manually evolving a concrete syntax editor based on the metamodel of a modeling language requires effort. Blended modeling proposes the application of and the seamless switching between multiple concrete syntaxes based on one metamodel, thereby increasing the effort through requiring realizing and evolving not one but multiple concrete syntax editors. To reduce this effort through automatization, the purpose of this deliverable is the description of two BUMBLE solutions regarding the generation of editors for the Eclipse technology ecosystem.

Table of Contents

BUMBLE Deliverable D3.4

Eclipse-based Blended Modeling Generation Environment Documentation 1
- Contributors 2
- Reviewers 2
- Project Acronyms 3
- Versions 3
- Executive Summary 4
- Table of Contents 5
- 1. Introduction 6
- 2. Editor Generation and Evolution Environment for EAST-ADL 6
  - 2.1. Generating Implementation Metamodels from Documentation Metamodels for the Evolution of EAST-ADL 7
  - 2.2. Post-processing of Initially Generated EATXT Grammars with the GrammarOptimizer for the Evolution of EAST-ADL 11
    - 2.2.1. Design of the GrammarOptimizer 12
    - 2.2.2. Use of the GrammarOptimizer 13
- 3. Automatic Generation of Graphical Editors from Annotated Ecore-based DSMLs 13
  - 3.1. Graphical Notation Specification Guide 15
  - 3.2. Model-to-text Transformations 17
  - 3.3. Discussion 18
- 4. Conclusion 18

1. Introduction

Modeling languages and their related metamodels are typically subject to evolution and evolving hand-crafted editors corresponding to the metamodels usually requires manual effort. An automatic approach for the realization of metamodel-based editors, via generative approaches, would markably reduce the effort of co-evolving editors (and models) in response to metamodel evolutions.

This deliverable describes BUMBLE solutions for the generation of editors in the Eclipse technology ecosystem. 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)”. Regarding the BUMBLE Technology Bricks and requirements (cf. deliverable D2.2), the work and solutions described in this deliverable contribute to the following ones:

Technology bricks	Description of main contributions	Main requirements
Editor Generators	We provide architectural descriptions for editor generators as part of different use cases in the Eclipse ecosystem.	BC1, BC2, BT1, BT2, BT3, BT4
(Meta-)model co-evolution	The editor generators mentioned above have the main purpose of supporting the language and thereby (meta-)model (co-)evolution.	BC9, BT22, BT24

From the BUMBLE use cases (UC) leveraging the Eclipse technology ecosystem (cf. deliverable D2.2), there are currently UC3 (Vehicular Architectural Modeling in EAST-ADL) and UC1 (Blended modeling in open-source) that can directly benefit from an editor generation approach. We describe an EAST-ADL editor generation and evolution environment for EAST-ADL in EATOP in Section 2, and an approach to generate graphical editors with Ecore and Sirius in Section 3.

2. Editor Generation and Evolution Environment for EAST-ADL

The Electronics Architecture and Software Technology – Architecture Description Language (EAST-ADL)[1] is a DSML for the specification of automotive embedded systems, which is used at Volvo Technology AB. The Eclipse-RCP-based tool suite EATOP[2,3] provides tree and form editors to enable the tool-based specification of EAST-ADL models based on the Eclipse Modeling Framework (EMF)[4]. In BUMBLE’s UC3 and w.r.t. the BUMBLE feature (B) (cf. Deliverable D2.2), we 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.

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

As the EAST-ADL language and thereby its metamodel is subject to evolution (the current EAST-ADL version is 2.2), maintaining the EATOP editors as well as the EATXT editor in a manual way is an effortful task. Thus, we describe in this section an editor generation and evolution environment for EAST-ADL. Figure 1 depicts the overall workflow for this environment.

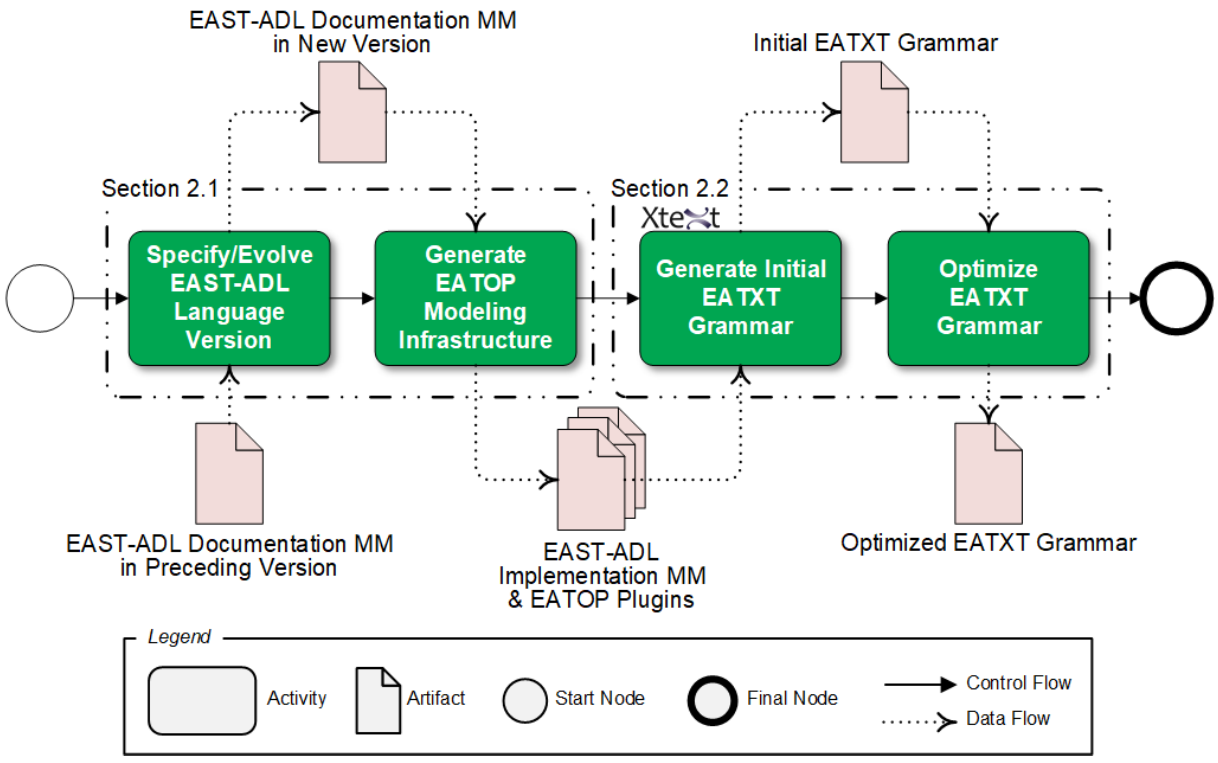

EAST-ADL Documentation MM in New Version
Initial EATXT Grammar
Section 2.1
Section 2.2
Xtext
Specify/Evolve EAST-ADL Language Version
Generate EATOP Modeling Infrastructure
Generate Initial EATXT Grammar
Optimize EATXT Grammar
EAST-ADL Documentation MM in Preceding Version
EAST-ADL Implementation MM & EATOP Plugins
Optimized EATXT Grammar
Legend
Activity
Artifact
Start Node
Final Node
Control Flow
Data Flow

Figure 1: Overall workflow for the EAST-ADL editor generation and evolution environment

The overall workflow starts with the specification/evolution of an EAST-ADL documentation metamodel, for which we provide means to generate the EATOP modeling infrastructure from it (described in Section 2.1). The second part of the overall workflow covers the generation of an initial EATXT grammar by means of the language engineering framework Xtext[5], for which we provide means to automatically optimize the initially generated grammar (described in Section 2.2).

2.1. Generating Implementation Metamodels from Documentation Metamodels for the Evolution of EAST-ADL[6]

Developing EATOP is itself a complex endeavor. The research project MAENAD[7] has addressed this complexity with an approach that distinguishes between a metamodel for documentation purposes

[5] https://www.eclipse.org/Xtext/
[6] This section contains contents of the MODELS'22 Tools & Demonstrations paper "Migrating from Proprietary Tools to Open-source Software for EAST-ADL Metamodel Generation and Evolution" (https://doi.org/10.1145/3550356.3559084).
[7] http://www.maenad.eu/

(*documentation MM*) and a structurally different implementation metamodel (*implementation MM*) that is deployed to EATOP. Particularly, the approach generates the structurally different implementation MM as well as other artifacts, such as customized model and edit code plugins for EMF, from the documentation MM. This generation of the EATOP modeling infrastructure decouples the model and edit code plugins specific to a concrete EAST-ADL version from the remaining EATOP plugins providing the actual tooling features. This means that EAST-ADL can evolve separately from EATOP.

However, this approach relies on the proprietary commercial-off-the-shelf (COTS) modeling tool SparxSystems Enterprise Architect[8], which annihilates the advantages and goals of the otherwise completely open-source-based metamodel evolution and generation workflow for EATOP. In the following, we report on our migrated Eclipse-native tooling for this workflow, which in comparison with the original approach based on Enterprise Architect supports rapid prototyping by eliminating tool gaps, avoids vendor lock-in, is platform-independent, and provides better maintainability.

Figure 2 depicts the general transformation approach from the EAST-ADL documentation MM to the implementation MM, with the EAST-ADL `Documentation MM` on the left-hand side and the EATOP `Implementation MM` on the right-hand side.

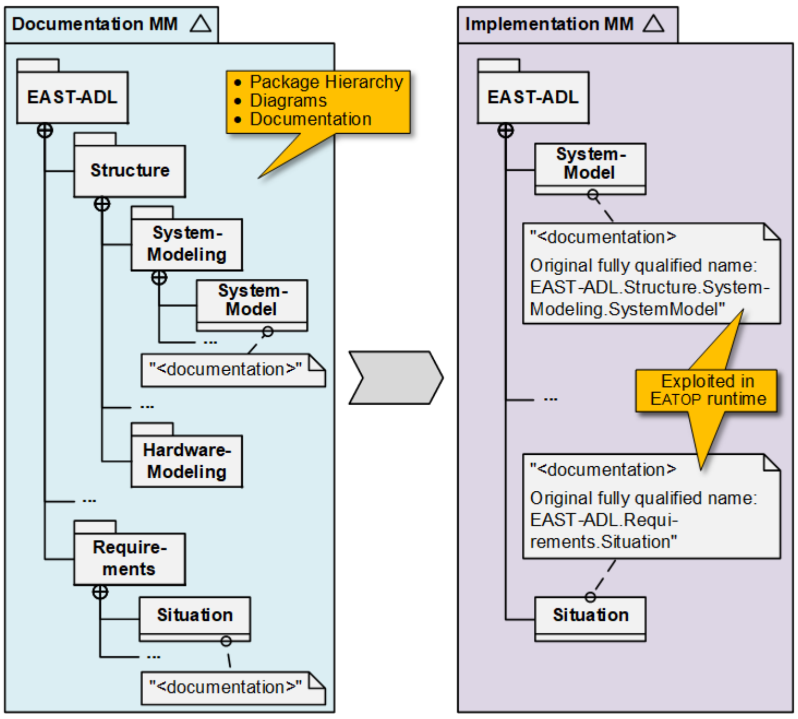


Figure 2: General approach for transforming
an EAST-ADL `Documentation MM` to an EATOP `Implementation MM`

The `Documentation MM` has the purpose of aligning the MM in its specific version with the corresponding version of the EAST-ADL specification. This encompasses structuring the MM into packages that correspond to the specification structure (e.g., the metaclass `SystemModel` in the

[8] https://www.sparxsystems.eu/enterprise-architect/enterprise-architect-editions

package `Structure.SystemModeling` and the metaclass `Situation` in the package `Requirements`), providing diagrams for each package, and providing documentation texts for the particular metaclasses (cf. annotations "`<documentation>`")—where the diagrams and documentation texts also show up in the specification.

The `Implementation MM` is the basis for generating model and edit code which applies a customized serialization format. This follows the standard approach used by EMF to generate a number of plugins which are deployed into the generic EATOP framework. To this end, the transformation approach flattens the package hierarchy from the `Documentation MM`. However, the original hierarchy is persisted by extending the original metaclass annotations with information about it. That is, all metaclasses (e.g., `SystemModel` and `Situation`) are rearranged directly under the root package EAST-ADL, and their documentation annotations are extended by their original fully qualified names (cf. annotations "`<documentation> Original fully qualified name: ...`"). Both the actual documentation and the original fully qualified name in these annotations are exploited at runtime in EATOP.

Figure 3 depicts the internal, technical workflow of our new Eclipse-native generator for the general transformation approach described in Figure 2. As in the original approach on Enterprise Architect, we realize the workflow in Java and apply the EMF and Sphinx[9] APIs. Similarly to the original workflow, the workflow is triggered by the evolution of the EAST-ADL metamodel and takes as input a documentation MM that is structured as explained in Figure 2. In contrast to the original workflow, this documentation MM is stored as an .ecore file and has been created and maintained using the documentation and diagramming features of EcoreTools included in EMF. In the step `Post-processing` (which is realized by means of an adapted subset of post-processing templates used in the original approach), the `Documentation MM` gets transformed into the `Implementation MM`. From this `Implementation MM`, the Eclipse plugins including a generator model, an EMF XMI schema, and the model/edit code are generated, like in the original approach.

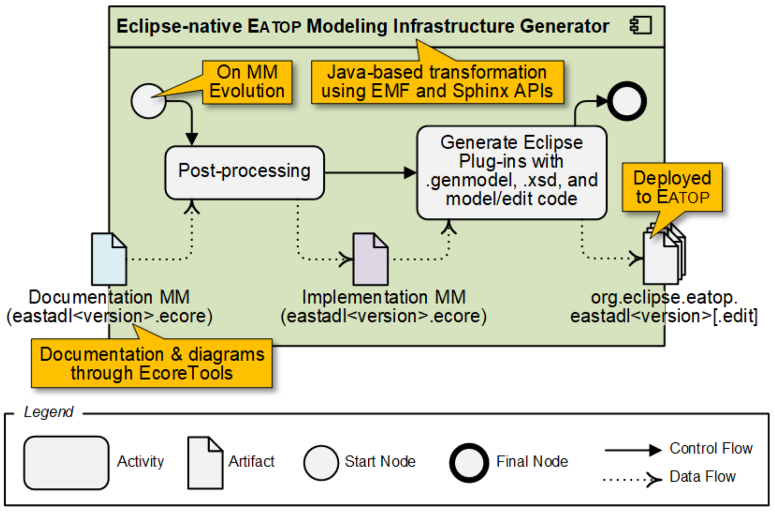
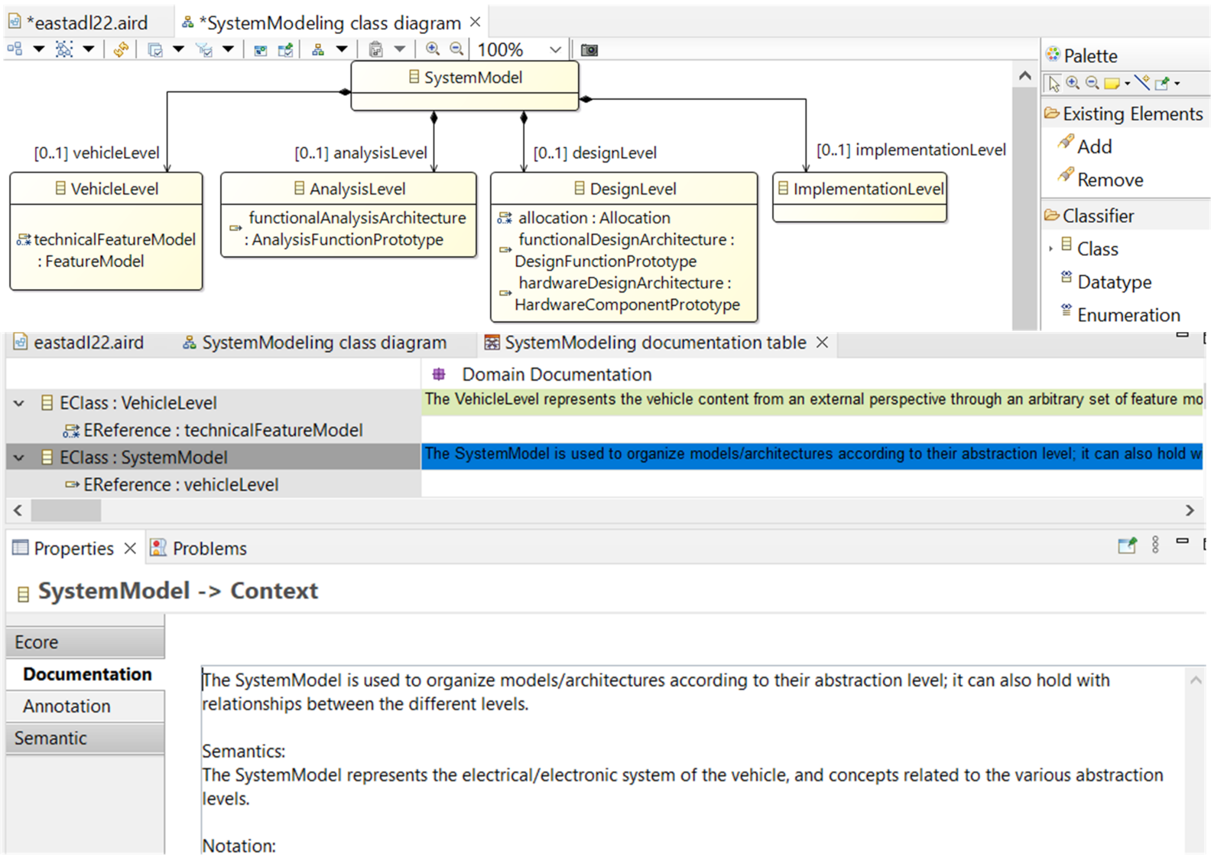

Eclipse-native EATOP Modeling Infrastructure Generator
On MM Evolution
Java-based transformation using EMF and Sphinx APIs
Post-processing
Generate Eclipse Plug-ins with .genmodel, .xsd, and model/edit code
Deployed to EATOP
Documentation MM (eastadl<version>.ecore)
Implementation MM (eastadl<version>.ecore)
org.eclipse.eatop.eastadl<version>[.edit]
Documentation & diagrams through EcoreTools
Legend
Activity
Artifact
Start Node
Final Node
Control Flow
Data Flow

Figure 3: New Eclipse-native workflow for the MM transformation

[9] https://www.eclipse.org/sphinx/

In the documentation MM, the textual documentation for the particular metaclasses and the package-wise diagrammatic documentation is conducted by means of the EMF-native component EcoreTools. Figure 4 depicts screenshots of this approach. The upper part of the figure shows a class diagram for the package `EAST-ADL.Structure.SystemModeling` (cf. left part of Figure 2). Like Enterprise Architect from the original approach and most other modeling tools, the EcoreTools diagram editor provides a model element palette, the actual diagram canvas with typical layouting capabilities, and a property editor. For the same package, the middle part of the figure depicts a documentation table, which provides a row-wise list of the package's metaclasses that can be selected for editing (i.e., metaclass `SystemModel` in the figure). In contrast to Enterprise Architect where the user has to open a property editor for each class, the documentation table provides a convenient overview of the documentation texts for all classes of a package. The bottom part of the figure shows the text editor for the selected table row. This text editor provides the same text editing capabilities as the property editor of Enterprise Architect.



The VehicleLevel represents the vehicle content from an external perspective through an arbitrary set of feature mo

The SystemModel is used to organize models/architectures according to their abstraction level; it can also hold w

The SystemModel is used to organize models/architectures according to their abstraction level; it can also hold with relationships between the different levels.

Semantics:
The SystemModel represents the electrical/electronic system of the vehicle, and concepts related to the various abstraction levels.

Notation:

Figure 4: Screenshots—package-wise class diagrams and class-wise documentation through EcoreTools

By eliminating the COTS tool Enterprise Architect from the generation and evolution workflow for the EAST-ADL language and metamodels, we establish a pure open-source solution and enable all open-source advantages, which were earlier annihilated by the incorporation of Enterprise Architect. The advantages encompass rapid prototyping, the avoidance of vendor lock-in, platform-independency, a better maintainability, equivalent documentation features, as well as the possibility of (meta-)model co-evolution.

We provide the new approach including a comprehensive user documentation as part of the EAST-ADL Bitbucket repository[10].

2.2. Post-processing of Initially Generated EATXT Grammars with the GrammarOptimizer for the Evolution of EAST-ADL

Xtext generates a default grammar from an Ecore metamodel, but the generated grammar is complex, not user-friendly, not easy to use, and enforces a very verbose style of writing. The default grammar, therefore, needs to be optimized to make it easier and user-friendly to users, and in line with the needs expressed by Volvo Technology AB in the BUMBLE Use Case 3. To this end, we first designed the grammar style in cooperation with Volvo, and modified the grammar to the one we want. But the modification work could be very cumbersome when the language is large. For example, EAST-ADL has more than 200 classes in its metamodel which results in about 3,000 lines of text in the Xtext file. It is cumbersome to do so many manual modifications. Therefore, we have to automate the modification work. To this end, we had two possible technical solutions to achieve it:

1. Customize the grammar generator of the Xtext framework, so that it could directly generate grammar in the appearance we want.
2. Modify the content of the *.xtext file after it is generated from the ecore metamodel.

The first possibility has the disadvantage that the grammar generator is dependent on the Xtext version used, because 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. Thus, it 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. So we chose the second solution.

Therefore, we designed the independent plug-in to realize the second potential solution. We called this independent plug-in GrammarOptimizer. The GrammarOptimizer takes the generated grammar file and transforms the style of the grammar by modifying the texts in the grammar file. Figure 5 shows the workflow for generating an optimized grammar with the second solution. The whole workflow process is:

1. Xtext generates the Xtext grammar (i.e., the initial textual grammar of a DSL) from the Ecore metamodel;
2. Configure the GrammarOptimizer according to the generated grammar and the domain concepts of the DSL (this step is operated by human manually);
3. Execute the GrammarOptimizer, i.e., the GrammarOptimizer optimizes the generated grammar according to the configuration automatically.

After those steps, we end up with an optimized grammar.

[10] https://bitbucket.org/east-adl/east-adl/src/Revison/org.eclipse.eatop/genmodelcodegen/

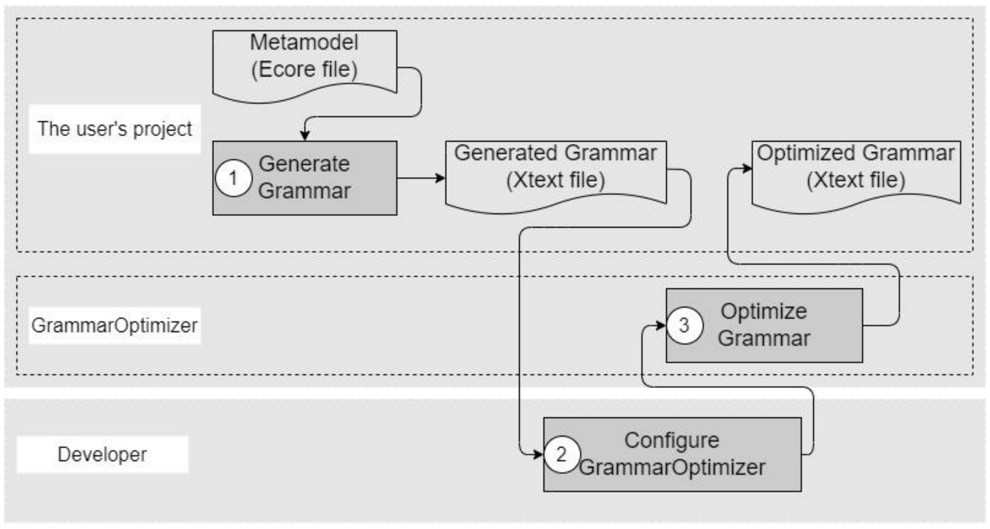


Figure 5: The workflow of generating a grammar with the second alternative solution

2.2.1. Design of the GrammarOptimizer

We designed GrammarOptimizer to parse an Xtext grammar into an internal data structure which is then modified and written out again. The language engineer controls this process via the API, usually in a standalone Java program that calls the necessary methods.

When GrammarOptimizer reads an Xtext grammar, it builds an internal data structure as depicted in Figure 6. A Grammar contains a number of GrammarRules that can be identified by their names. The rule in turn consists of a sorted list of LineEntrys with their textual lineContent and an optional attrName that contains the name of the attribute defined in the line.

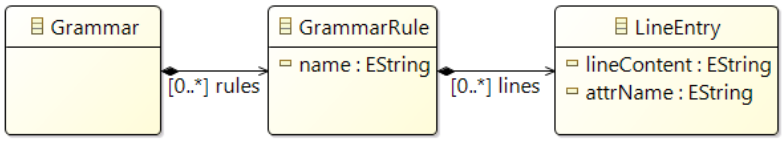


Figure 6: The class design for grammar rules.

Optimization rules are all derived from the abstract class OptimizationRule as shown in Figure 7. Derived classes overwrite the apply() method to perform the specific text modifications for this rule. In doing so, the specific rule can access the necessary information through the class members: field grammar (i.e., the entire grammar), grammarRuleName (i.e., the name of the specified grammar that a user wants to optimize separately), attrName (i.e., the line that a user wants to optimize exclusively). Sub-classes can also add additional members if necessary.

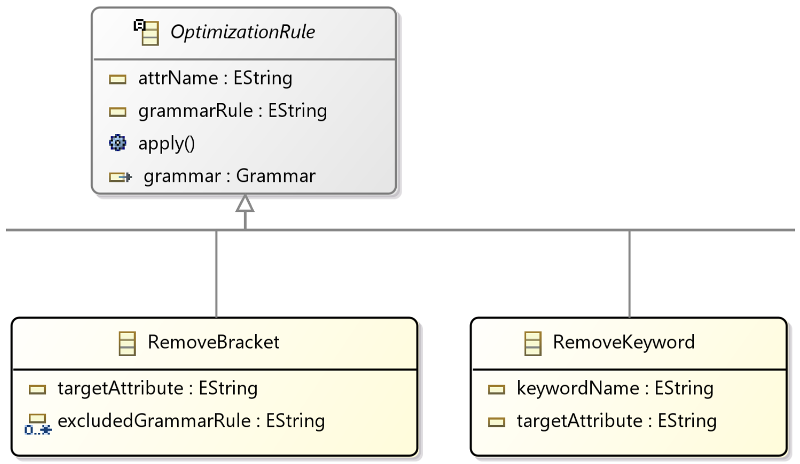


Figure 7: Excerpt of the class diagram for optimization rules.

2.2.2. Use of the GrammarOptimizer

The GrammarOptimizer is an independent Eclipse plugin implemented in Java. Before using it, you need to import it into Eclipse, and you should make sure that its code is in the workspace where the project `org.bumble.eatxt` is located. Then the steps are:

1. Right-click on the plugin project (i.e., `org.bumble.xtext.grammaroptimizer`).
2. Click “run as” and then click “Java Application”

Optionally, if you want to start the optimization 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 `EAXML` as entry rule.
    b. Specify `org.bumble.eatxt` as project name, `org.bumble.eatxt.Eatxt` as language name, and `eatxt` as extensions.

Once you have optimized the grammar with the GrammarOptimizer, the modeling workflow engine file (i.e., *mwe2 file) has to be run to generate the Xtext artifacts (i.e., the editor) of the language. This completes the workflow steps to generate the textual editor for EAST-ADL.

3. Automatic Generation of Graphical Editors from Annotated Ecore-based DSMLs

Sirius[11] is an open-source Eclipse Project that allows the creation of graphical modeling workbenches leveraging the Eclipse Modeling Framework (EMF) and Graphical Modeling Framework (GMF)[12]. Relying on EMF facilitates the integration of Sirius with a variety of modeling technologies based on EMF such as Xtext, a framework for the development of textual domain-specific languages. Thus, the fact that both Sirius and Xtext are based on EMF, enables a basic and out-of-the-box integration that

[11] https://www.eclipse.org/sirius/overview.html
[12] https://www.eclipse.org/modeling/gmp/

allows for seamless switching and synchronization between graphical and textual editors. The process of integrating Sirius and Xtext is explained in the following.

The overall workflow starts by defining the language in a grammar using Xtext. The XtextResource contains a parser that allows for the generation of an Ecore model from the grammar, where the Ecore model can be used by other EMF based tools such as Sirius. Given the Ecore model, Sirius allows to specify representations by creating and configuring a Viewpoint Specification Model (VSM) which defines and configures viewpoints and their associated representations (e.g., diagrams). However, creating the VSM in a manual manner can be challenging for new users and a tedious, repetitive, and error-prone process for experienced users. To simplify this task, we analyze how to describe a graphical notation to be used as input to automate the generation of Sirius graphical editors from it.
As part of the solution, we have developed i) a graphical notation specification guide for annotating Ecore models and describing each element's concrete syntax and ii) model-to-text transformations (M2T) that can generate VSMs containing the *.odesign file extension.

Figure 8 illustrates the steps that a user must take in order to generate a Sirius graphical editor using our proposed approach. Assuming that the Ecore model has already been defined and validated, the user must complete the following steps.

1. Annotate the Ecore model using the graphical notation specification guide. The outcome is an annotated Ecore model.
2. Validate the annotated Ecore model. If the validation fails, the user needs to re-annotate the Ecore model (back to Step 1); alternatively, if the validation is successful the user creates a new runtime instance and proceeds with the next step.
3. Run M2T transformations. The outcome is an *.odesign file.
4. Generate the graphical editor using the *.odesign file. The outcome is a Sirius graphical editor.
5. Validate the graphical editor. If the validation is successful, the outcome is a fully functioning graphical editor; alternatively, the user has to re-annotate the Ecore model (back to Step 1).

In Section 3.1 we describe the graphical notation specification guide, and in Section 3.2 we provide implementation details on M2T transformations. Lastly, in Section 3.3 we detail a discussion of the proposed approach.

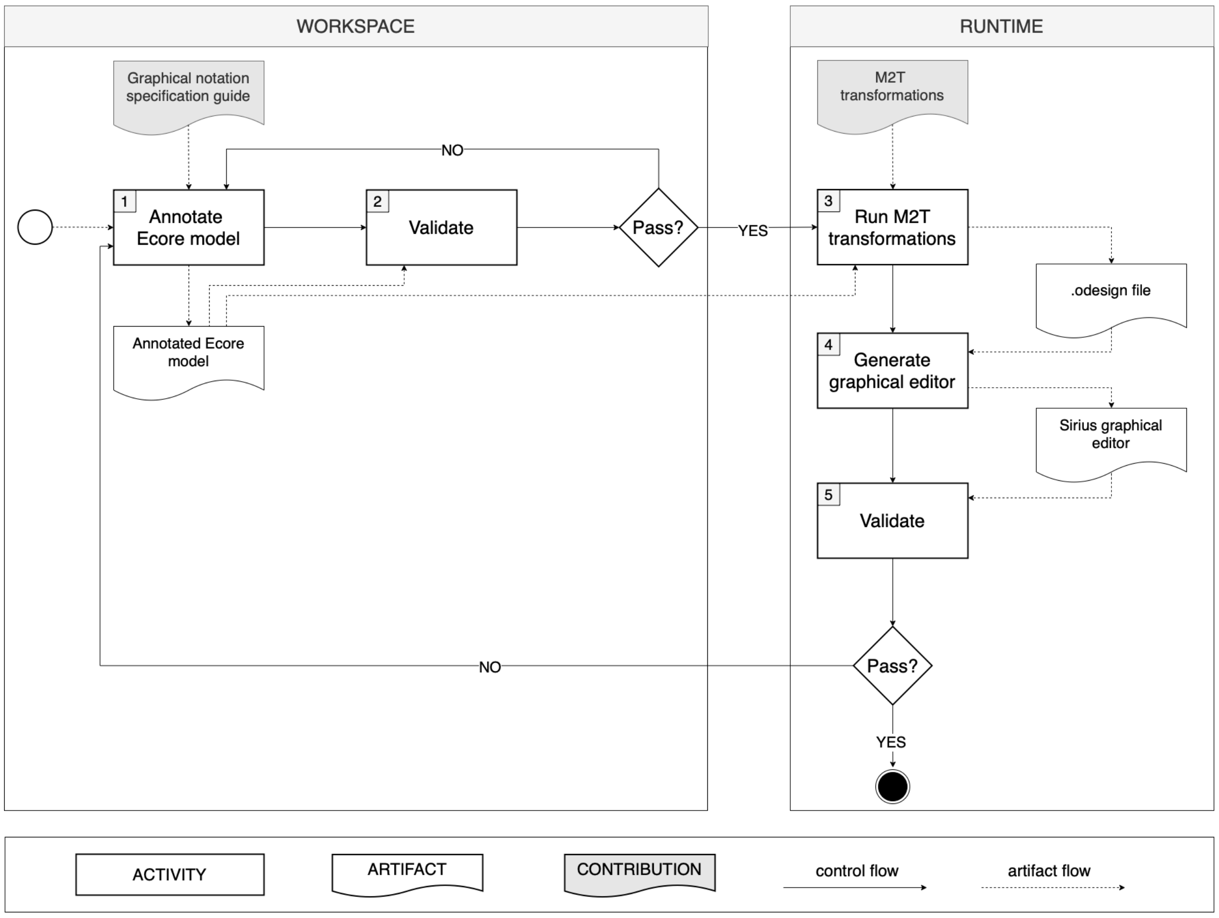


Figure 8: Automatic generation of Sirius graphical editors

3.1. Graphical Notation Specification Guide

The graphical notation specification guide is a crucial element of the proposed solution that allows the user to define the graphical representation for each concept of the underlying metamodel. To define the graphical notation specification guide, we first identified a set of concepts that are frequently used during the creation of graphical editors and incrementally added more relevant concepts. These concepts can be described as EAnnotations, that can be used to annotate EMF models/metamodels and describe how each concept of the latter should be visualized in the generated graphical editor. The most common concepts that we identified are *diagrams, containers, nodes,* and *edges* which are part of the majority of graphical editors. We want the graphical editor to allow both the visualization and editing of the representations. Thus, we focus not only on EAnnotations that will allow users to visualize the elements, but also on EAnnotations that will allow the user to create, edit and delete elements. In the following, we detail the graphical notation specification guide, by presenting the main concepts, providing a description of the concepts, and giving an example of how the user can define the annotations and the required details.

@sirius - Used to define the viewpoint naming and diagram naming, the root of the metamodel, and the file extension of the metamodel.

```
@sirius(name = "", mainclass="", viewpoint="", diagram="")
```

@namespace - Used to define the URI of the metamodel and the prefix for the models instantiated from the metamodel.

```
@namespace(uri="", prefix="")
```

@sirius.container - Used for EClasses that should appear on the diagram as containers. A container is an element that can contain other elements.

```
@sirius.container(name="", expression="", background="", foreground="", container="", domain="", icon="", presentation="", bordercolor="", mapping="")
```

@sirius.node - Used for EClasses that should appear on the diagram as nodes. A node is an element that cannot contain other elements.

```
@sirius.node(name="", expression="", style="", label="", icon="", color="", container="", height="", width="", domain="", labelcolor="", bordercolor="", icon="")
```

@sirius.altnode - Used to specify an alternate style for a node, where the alternate style is applied based on a boolean condition.

```
@sirius.altnode(belongsto="", condition="", color="", label="")
```

@sirius.borderednode - Used for nodes which appear on the side of other elements (e.g., ports for state machines).

```
@sirius.borderednode(name="", expression="", style="", color="", label="", icon="")
```

@sirius.edge - Used for EClasses that should appear on the diagram as connections between diagram elements.

```
@sirius.edge(type="", source="", target="", containerreuse="", color="", domain="" , name="", label="", targetFinderExpression="", sourceFinder Expression="", targetStyle="", sourceStyle="", expression="")
```

@sirius.altedge - Used to specify an alternate style for an edge, where the alternate style is applied based on a boolean condition. This alternate style may specify another color or label to the edge and is mainly used to visualize an internal attribute that would otherwise not be visualized.

```
@sirius.altedge(belongsto="", condition="", color="", label="")
```

@sirius.decoration - Used for adding small graphical annotations to the diagram elements.

```
@sirius.decorations(name="", type="", style="", expression="", image="", mappings="")
```

@sirius.tool - The tools that enable editing of elements in the graphical editor can all be defined by using the **@sirius.tool** EAnnotation. To differentiate between their capabilities we use the `type` attribute as follows.

- **Create** - Used to create elements in the graphical editor.
  `@sirius.tool(name="", type="CreationDescription", mappings="", reference="", belongsto="", source="", target="", containerexpression="")`
- **Delete** - Used to delete elements in the graphical editor.
  `@sirius.tool(name="", type="DeleteElementDescription", feature="", mappings="", belongsto="")`
- **Edit** - Used to edit the labels of elements directly in the diagram itself, by clicking on the label.
  `@sirius.tool(name="", type="DirectEditLabel.", feature="", mappings="", belongsto="")`

- **Redirect** - Used to redirect edges from one element to another.
  `@sirius.tool(name="", type="ReconnectEdgeDescription", feature="", mappings="", belongsto="")`

3.2. Model-to-text Transformations

M2T transformations enable the generation of Sirius graphical editors by transforming the annotated Ecore model that contains details related to both abstract and concrete syntax into VSMs (i.e., *.odesign file). For the definition of M2T transformations, we used Xtend[13], an open-source, expressive, and flexible language whose IDE is integrated within the Eclipse IDE. It provides templates for code generation and is commonly used for M2M and M2T transformations.

The transformations take as input the annotated metamodel and generate the XML representation of the *.odesign file. The defined template of the transformations are designs to generate XML files following the structure of *.odesign files. The generated *.odesign file can then be used to generate a graphical editor for the respective metamodel. The basic idea is that the M2T transformations iterate through the metamodel concepts, fetch the corresponding information described in the annotations for each concept, and generate text conforming to XML to populate the *.odesign file. In Listing 1, we provide an example of the compile function which is the starting point of the implementation. The text in between triple quotes is printed as plain text, and the text in between << >> is evaluated and the outcome is printed. As it can be seen it follows the structure of an XML schema file defining the XML version and encoding. The `mmPackage` represents the metamodel package and functions such as `getViewpoint(mmPackage)` and `getDiagram(mmPackage)` iterate through the annotations of the `mmPackage` and collect the required information.

Listing 1: `compile` function of the M2T transformations

```
public def compile(Package mmPackage)
        '''<?xml version="1.0" encoding="UTF-8"?>
        <<val viewpoint = getViewpoint(mmPackage)>>
        <<val diagram = getDiagram(mmPackage)>>
                <<val fileextension = getExtension(mmPackage).toLowerCase()>>
                <description:Group xmi:version="2.0" ...>
                <ownedViewpoints name="<<viewpoint>>"
              modelFileExtension="<<fileextension>>">
                <<val mainclass = getMainClass(mmPackage)>>
                <ownedRepresentations xsi:type="description_1:DiagramDescription" name="<<diagram>>"
domainClass="<<mmPackage.name>>::<<mainclass>>">  <metamodel href="<<mmPackage.nsURI>>"/>
                            <defaultLayer name="Default">
                ...
                        </defaultLayer>
                        </ownedRepresentations>
                        <ownedJavaExtensions qualifiedClassName="<<getPath(mmPackage)>>.Services"/>
                        </ownedViewpoints>
                </description:Group>
                '''
```

[13] https://www.eclipse.org/xtend/

3.3. Discussion

The proposed approach was validated utilizing three use cases that implement complementary features. We used the basic family metamodel, the UML-RT for state machines metamodel and the NoSQL visualization tool metamodel. The graphical notation specification guide and M2T transformations were iteratively revised whenever errors were identified in the generation of a particular editor. Additionally, the validation involved inspecting whether the resolved errors affected the generation of other editors. Lastly, for any missing concepts in our editor, we enriched the graphical notation specification guide, and the M2T transformations to include them. The metamodels and the complete implementation can be found online[14].

The solution can reduce the effort involved in the process of creating graphical editors and co-evolving graphical editors in response to metamodel evolution. In the case of creating graphical editors in Sirius from scratch, the approach of including all information relevant for creating a graphical editor in Sirius in one single file requires less effort, is less error-prone, and more intuitive. Same argument is true for co-evolving graphical editors. For instance, consider that the user wants to make changes to the underlying metamodel. With the manual approach, the user must make the changes to the metamodel in the workspace instance, register it, and then import it in the runtime instance. Then, the user must revise the *.odesign file to reflect the changes made in the metamodel by using the tree editor offered by Sirius out-of-the box. Alternatively, using our proposed approach the user can make the changes to the metamodel and annotations that describe how the concepts should be graphically represented, in one single textual or tree editor, which allows for increased usability. In summary, this makes for an easier way of creating and co-evolving graphical editors, thus facilitating the tedious and repetitive manual process.

4. Conclusion

In this deliverable, we presented two approaches for automatically generating editors based on language metamodels in the Eclipse ecosystem. These generative approaches markably reduce the engineering effort in comparison to hand-crafting modeling editors in the case of language evolution.

First, in Section 2, we presented the editor generation and evolution environment for EAST-ADL, which generates from a documentation metamodel both EATOP editors and an implementation metamodel. Based on this implementation metamodel, we use Xtext to initially generate a grammar and apply our GrammarOptimizer to automatically post-process this grammar in order to yield the EATXT editor. Then, in Section 3, we presented an approach to generate graphical editors from annotated metamodels. In this context, an arbitrary Ecore metamodel is annotated following a graphical notation specification guide. From the annotated metamodel, a Sirius configuration is generated that can subsequently be used as concrete graphical editor for instances of the Ecore metamodel.

[14] https://github.com/JocksTheGul/Auto-generated-graphical-editors-in-Sirius