Exploiting Meta-Model Structures in the Generation of Xtext Editors

Exploiting Meta-Model Structures in the Generation of Xtext Editors

Jörg Holtmann[1][a], Jan-Philipp Steghöfer[2][b] and Weixing Zhang[1][c]
[1]*Department of Computer Engineering, Chalmers University of Technology | University of Gothenburg, Sweden*
[2]*Xitaso IT & Software Solutions GmbH, Augsburg, Germany*
{*jorg.holtmann, weixing.zhang*}*@gu.se, jan-philipp.steghoefer@xitaso.com*

Keywords: Xtext, textual modelling, modelling language engineering, modelling environments

Abstract: When generating textual editors for large and highly structured meta-models, it is possible to extend Xtext's generator capabilities and the default implementations it provides. These extensions provide additional features such as formatters and more precise scoping for cross-references. However, for large metamodels in particular, the realization of such extensions typically is a time-consuming, awkward, and repetitive task. For some of these tasks, we motivate, present, and discuss in this position paper automatic solutions that exploit the structure of the underlying metamodel. Furthermore, we demonstrate how we used them in the development of a textual editor for EATXT, a textual concrete syntax for the automotive architecture description language EAST-ADL. This work in progress contributes to our larger goal of building a language workbench for blended modelling.

1 Introduction

Xtext (Eysholdt and Behrens, 2010) is a framework for the development of domain-specific languages (DSLs). It can either take an existing meta-model and derive a grammar from it or allows a language engineer to create a grammar directly which is then translated into a meta-model. Once a grammar exists, Xtext can generate editors that integrate seamlessly into the Eclipse IDE and offer many convenient features such as an outline view of the file which is currently edited. On the other hand, it offers extension mechanisms for more advanced editor features.

In practice, using these extensions mechanisms can pose significant technical challenges. For example, auto-formatting or the use of template proposals — both common features in modern editors — are not supported for DSLs based on Xtext out-of-the-box. Despite comprehensive documentation of the corresponding extension mechanisms, these challenges re-occur and have to be solved manually. In particular, implementing these features for sufficiently large languages can be cumbersome and involves a lot of repetitive code.

In other cases where Xtext provides support out-of-the-box, the default implementations provided by Xtext are not always suitable for large DSLs since the performance they provide (e.g., for cross-reference auto-completion) is insufficient for practical purposes or for certain use cases.

In the context of a prototype for a textual variant of the automotive systems modeling language EAST-ADL (Debruyne et al., 2004; Cuenot et al., 2007), we implemented several automated solutions for these re-occurring challenges by using custom *Xtext generator fragments* that exploit the structure of the language's meta-model. These fragments are used when Xtext generates the editors for a DSL. We created fragments for formatting, content-assist, and template proposals. In addition, we created a solution for providing the scope of cross-references automatically suggested by the editor which addresses several short-comings of the default solution. In all cases, the structure of the meta-model provides the necessary support to enable the automation.

In this position paper, we discuss these automated solutions and describe how other DSL engineers can adapt them for their needs. Our solutions are particularly suitable for very large, but highly structured DSLs that are available as a metamodel, but should also translate to other situations in which Xtext is used. Furthermore, we discuss the limitations of the automatic solutions where present. Whereas these automations are not generic enough to add them to Xtext properly, they are interesting for other language engi-

[a] https://orcid.org/0000-0001-6141-4571
[b] https://orcid.org/0000-0003-1694-0972
[c] https://orcid.org/0000-0003-2890-6034

neers and can be applied to other languages as well.

Our work in progress contributes to a larger vision of a language workbench for blended modelling in which editors for different concrete syntaxes for the same abstract syntax co-exist and enable engineers to seamlessly switch between the representations (Ciccozzi et al., 2019). Especially when used in conjunction with evolving languages where the meta-model structure changes regularly, it is necessary to be able to quickly regenerate the editors with as little manual effort as possible.

2 Related Work

Neubauer et al. proposed an approach (Neubauer et al., 2015) to automatically create modern editors for XML-based DSLs by bridging the "technical spaces" *XMLware*, *grammarware*, and *modelware*. Their automation mainly focuses on providing automatic customization of textual concrete syntaxes for the target DSL, such as which symbols to use. In contrast, our automation mainly focuses on the enhancement of the editors, such as automatically providing default names for new elements, improved suggestions for cross-references, etc.

Latifaj et al. focused on enabling different stakeholders to perform blended modeling (David et al., 2022) in an automated manner, i.e., using different modeling notations to seamlessly handle overlapping parts of the model (Latifaj et al., 2021). They discuss four challenges to achieving this automation goal in the commercial tool RTist. However, they do not address the challenges of content assistance, template proposals, etc., which are the core challenges discussed in this paper.

Cooper and Kolovos acknowledge some of the challenges we address in this work in their paper on requirements and challenges of blended modelling (Cooper and Kolovos, 2019). For instance, they consider scoping across concrete syntaxes to be a challenge. We address this with our custom ScopeProvider as discussed in Section 4.4.

3 Background

In the following, we provide relevant information on Xtext as well as EATXT, the textual variant of EAST-ADL which we used as an exemplar and a case study for our work.

3.1 Xtext

Xtext is a framework for the development of textual domain-specific languages (DSL) (Eysholdt and Behrens, 2010). At its core is a grammar language that allows defining the syntax of a DSL. Xtext can either generate a grammar out of an existing meta-model or the user can specify the grammar directly and the meta-model is generated by Xtext. A grammar can be used as the input to generate editors for the Eclipse IDE or for browsers. In addition, Xtext can generate a language server that allows integrating the DSL into VS.Code or Eclipse Theia.

The generation process is controlled by a workflow for the Modeling Workflow Engine (MWE2) (Eclipse Foundation, 2023b). Xtext provides a *language generator* which can be customized and extended with custom *fragments*. A fragment generates code based on the generator's configuration, the grammar and the corresponding meta-model. Xtext out-of-the-box provides a number of such fragments, which can be extended or replaced. These fragments add a number of features to the generated editors. For instance, Xtext automatically generates a syntax validator which highlights incorrect syntax directly in the editor and in Eclipse's "Problems" view. For editors in the Eclipse IDE, support for the outline view is also generated. We use Xtext's ability to change the standard configuration to add custom fragments that provide better formatting, content-assist and template proposals as described below. These custom fragments are written in Xtend (Eclipse Foundation, 2023a).

Xtext splits the generated code into different bundles, distinguishing between the infrastructure for the language itself, the user interface of the editors (bundle name ends with ".ui") and the integration into the Eclipse IDE (bundle name ends with ".ide").

3.2 EAST-ADL and EATXT

EAST-ADL is an automotive systems modeling language (Cuenot et al., 2007) and is based on a large metamodel with more than 200 metaclasses and a hierarchy of nested elements describing different aspects of electronic vehicle systems. It can be edited in EATOP, an Eclipse-based editing environment that provides a hierarchical view of an EAST-ADL model along with form- and table-based editing capabilities. EATXT provides a textual syntax for EAST-ADL with the goal to enable blended modeling (Ciccozzi et al., 2019), that is, the ability to switch between the hierarchy-based and the textual representation seamlessly depending on the editing task.

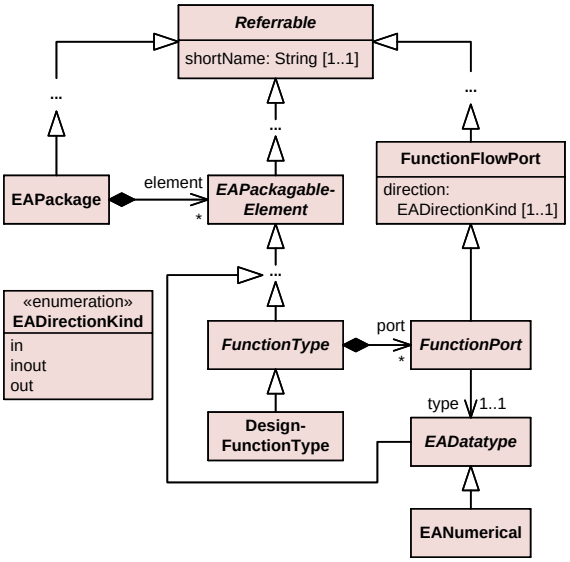

Figure 1: EAST-ADL Metamodel Excerpt

Figure 1 depicts an EAST-ADL metamodel excerpt, which we use in this paper as a running example for illustrative purposes. The excerpt contains a set of metaclasses (partially containing attributes) and relationships (i.e., generalizations, compositions, and cross-references), which we explain in the following.

An example of an EATXT file is shown in Figure 2. The textual concrete syntax follows the hierarchy and structure of the EAST-ADL metamodel. Metaclasses in EAST-ADL are represented as blocks delimited by curly braces (e.g., the FunctionFlowPort `WipingCmd` in lines 7–10 as part of the DesignFunctionType `WiperCtrl`). Attributes are represented as lists with the attribute name and the values (e.g., the attribute direction in line 8 as part the FunctionFlowPort `WipingCmd`). Cross-references are represented as the reference name followed by the actual path to the reference (e.g., the cross-reference type points to "`DataTypes.Integer_uint8`"). In the case of EATXT, the Xtext grammar is specified in such a way that the textual keywords representing the metamodel concepts like metaclasses, attributes, and cross-references are named as in the metamodel (e.g., both the keyword and its corresponding metaclass have the same name FunctionFlowPort).

```
1  EAPackage DataTypes {
2      EANumerical Integer_uint8
3  }
4
5  EAPackage FcnDesignTypes {
6      DesignFunctionType WiperCtrl {
7          FunctionFlowPort WipingCmd {
8              direction out;
9              type "DataTypes.Integer_uint8";
10         }
11     }
12
13     DesignFunctionType WiperMotor {
14         FunctionFlowPort WipingCmd {
15             direction in;
16             type "DataTypes.Integer_uint8";
17         }
18     }
19 }
```

Figure 2: Excerpt from an EATXT File Specifying a Windshield Wiper Control System

As described in Section 3.2, EAST-ADL is a stable language based on an Ecore metamodel (cf. eastadl22.ecore in the plugin o.e.eatop.eastadl22 in the top-left corner of Figure 3). We conceived its textual syntax by automatically deriving an initial grammar from that metamodel with Xtext and subsequently adapting the grammar to the stakeholders' needs. Figure 3 indicates the resulting grammar as the artifact Eatxt.xtext as part of the plugin org.bumble.eatxt. In this grammar, we named the production rules and keywords equally as the language concepts in the metamodel (i.e., metaclasses, attributes, and EReferences). This enables the exploitation of the metamodel structure and thereby the development of our automatic solutions.

The green Xtend and Java classes that are part of the plugin org.bumble.eatxt represent our solutions to the re-occurring challenges described below, exploiting the structure of the metamodel eastadl22.ecore. Three of the solutions are custom fragments that are executed by the Xtext language generator (cf. Section 3.1) and generate further artifacts (depicted in lilac) that are used in the EATXT runtime. Another solution generates an artifact during the activation of the plugin which can then be exploited at runtime. We provide the implementation in our EATXT development repository (Holtmann et al., 2023). In the following four subsections, we explain and discuss these automatic solutions, the challenges they solve, and the artifacts they generate.

4 Challenges and Solutions

In this section, we explain our solutions to re-occurring challenges in the development of Xtext-based language workbenches by referring to Figure 3, which depicts the coarse-grained architecture of our Xtext-based editor for EATXT.

4.1 Template Proposals

The Eclipse IDE provides the possibility of using *code templates* to enter complex code constructs that contain a significant amount of text and are more

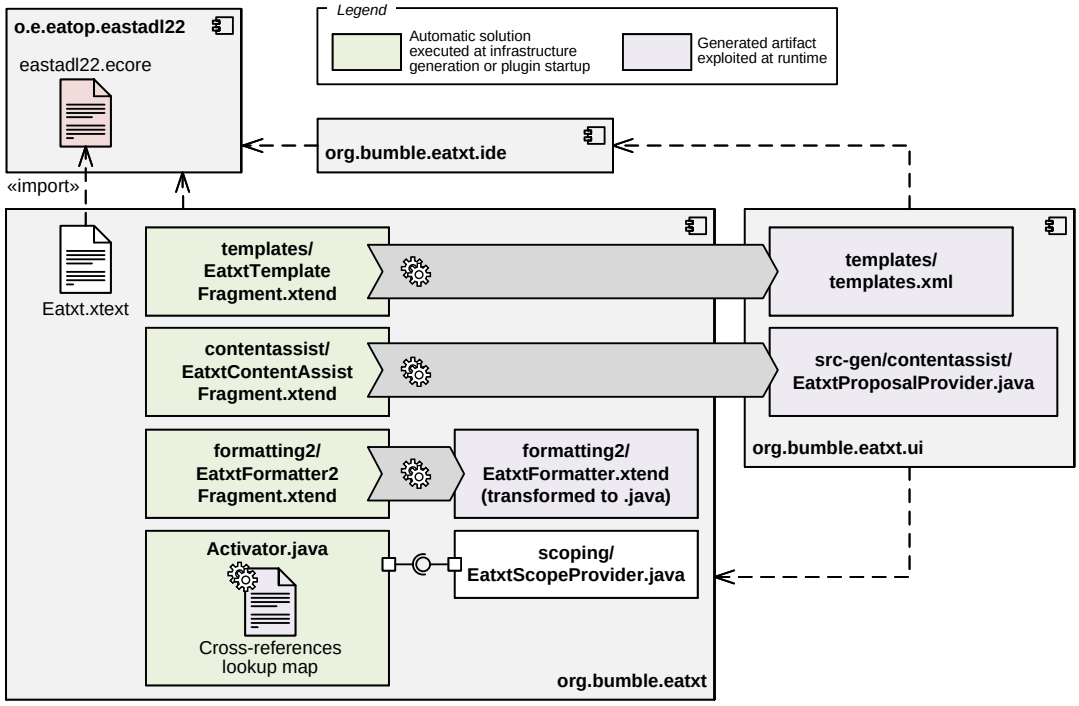

Figure 3: EATXT Architecture and Automatic Solutions for Re-occurring Challenges

complicated than simple keywords. For instance, in the case of the Java programming language, the proposal of complete constructors or different kinds of loops is handled by code templates. Eclipse ships with a set of pre-defined code templates and allows adding user-defined templates or customize existing ones. Likewise, Xtext editors also provide the possibility to use and define code templates by supporting *template proposals*, which consist of the actual code template and a context type (Eclipse Foundation, 2023g). During the editor generation, Xtext automatically registers such a context type for each production rule and keyword (e.g., the production rule for textual instances of the EAST-ADL metaclass FunctionFlowPort), and it provides the context for the code template proposal.

However, the development of template proposals is cumbersome, because the DSL engineer has to define each template manually in an XML file templates/templates.xml as part of the UI plugin (cf. top of the plugin org.bumble.eatxt.ui on the right-hand side of Figure 3) (Eclipse Foundation, 2023g). Moreover, it is recommended practice to define them in the runtime workspace of the Xtext editor through a corresponding dialog and to export the resulting templates.xml to the development workspace (Eclipse Foundation, 2023g), which impedes rapid prototyping of the template proposals. Particularly, this manual practice is awkward for large metamodels.

As an automatic solution to this challenge in EATXT, we implemented the generator fragment templates/EatxtTemplateFragment.xtend as part of the plugin org.bumble.eatxt (cf. fragment of the plugin in Figure 3). This fragment is executed by the MWE2 workflow and automatically generates the XML file templates/templates.xml as part of the plugin org.bumble.eatxt.ui. The fragment iterates over the metamodel and generates a template proposal for all metaclasses and all of their mandatory sub-elements (i.e., attributes, containments and their nested structures, as well as cross-references). We restrict the code templates to contain only sub-elements that are mandatory in the metamodel instead of proposing all potential sub-elements, because it might be much effort for the user to delete all proposed optional, but potentially not required, elements.

Figure 4 depicts the generation scheme of the fragment using the example of the metaclass FunctionFlowPort and its mandatory attribute and mandatory cross-reference. We specify the context type as the name of the production rule, which in the EATXT case is always named the same as the corresponding metaclass (e.g., org.bumble.eatxt.Eatxt.FunctionFlowPort for the XML template attribute context). The actual code template is then generated as the name of the production rule and metaclass (e.g., FunctionFlowPort) followed by a set of further texts as well as opening and closing curly braces.

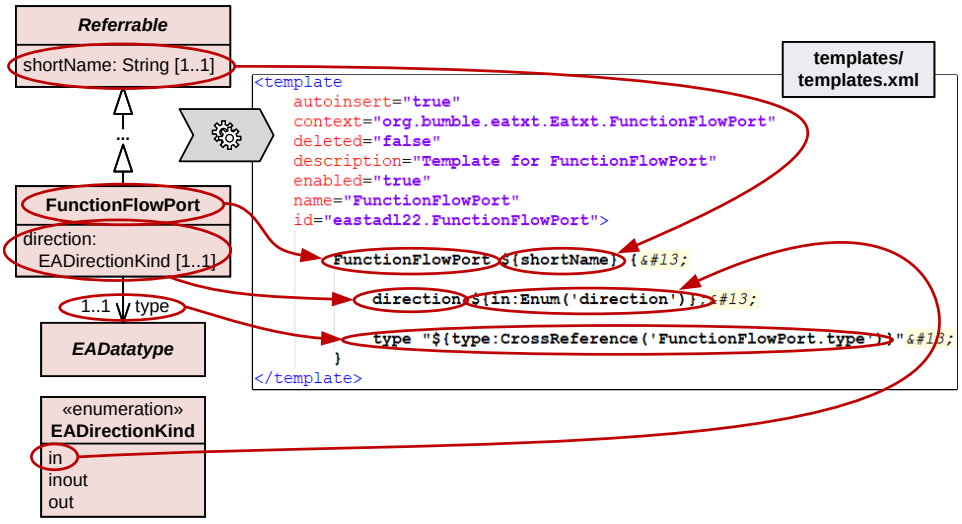

Figure 4: Exemplary Generation of a Template Proposal

Beyond proposing simple static texts that would typically not fit to the remainder of the text file and hence would lead to error messages in the editor, the template proposal approach also supports *template variable resolvers* (Eclipse Foundation, 2023g). For simple attributes, we distinguish the different plain data types of attributes (e.g., String, Integer, Float) and corresponding template variable resolvers. For example in Figure 4, we translate the String attribute shortName of the metaclass Referrable to a corresponding template variable resolver ${shortName}. For enumeration attributes, we generate enumeration template variable resolvers. For example, the value of the attribute direction is automatically set to the first literal in of the enumeration EADirectionKind. Furthermore, we also support cross-reference template variable resolvers that propose an existing element that fits to the type of the cross-reference. For example, the cross-reference type is translated to the corresponding resolver.

Figure 5 depicts a screenshot for the proposal of a code template for the context type FunctionFlowPort with its mandatory two attributes, where the default enumeration value of the direction attribute as well as a target candidate for the cross-reference type is directly proposed. In the case of EATXT, the generated template file contains 194 code templates with a total of more than 1,000 XML lines and covers all metaclasses of the metamodel and their mandatory subelements.

As we exploit the concept names of the underlying metamodel for this solution, such a generation of template proposals is restricted to grammars that have the same concept names as the corresponding metamodel (i.e., metaclass names, attribute names, association role names). Thus, the DSL engineer is not allowed to rename the resulting keywords.

4.2 Content-assist for new Model Elements with Unique Names

Beyond the usual keyword-based content-assist known from IDEs, Xtext provides a customizable approach to provide content-assist for the specification of new model elements by means of *proposal providers* (Eclipse Foundation, 2023c). In this approach, Xtext generates an abstract proposal provider class with default functionality with content-assist methods for all metaclasses as well as an empty concrete subclass that can be customized. To customize the content-assist for a specific model element type (e.g., for the metaclass FunctionFlowPort with exemplary instances in lines 7–10 and 14–17 in Figure 2), the DSL engineer has to override the corresponding method in a concrete subclass.

In the case of EATXT, we had the requirement to provide content-assist proposals with unique names in the model namespace for all metaclasses with a mandatory shortName attribute. Since almost all of the more than 200 metaclasses in the EAST-ADL metamodel have this mandatory attribute due to subclassing (cf. Figure 1), the implementation of the corresponding particular methods in the concrete proposal provider would have been very repetitive.

In order to provide an automatic solution for this challenge, we implemented a fragment contentassist/EatxtContentAssistFragment.xtend (cf. plugin org.bumble.eatxt in Figure 3). Again, this fragment is executed by the MWE2 work-

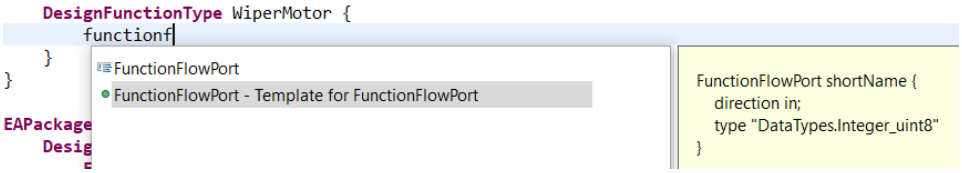

Figure 5: Proposed Code Template

flow and exploits the metamodel structure by iterating over all metaclasses with the mandatory attribute and generating the concrete subclass src-gen/contentassist/EatextProposalProvider.java (cf. plugin org.bumble.eatxt.ui on the right-hand side of Figure 3).

Figure 6 depicts the scheme for the generation of the proposal provider methods using the example for the metaclass FunctionFlowPort. For each metaclass with the mandatory attribute, this generated proposal provider includes a corresponding overriding content-assist method that proposes a unique name consisting of the corresponding prefix <metaclassName_> followed by a randomized number for a new instance of the metaclass. For example, we generate for the metaclass FunctionFlowPort a corresponding proposal provider method that proposes the prefix "FunctionFlowPort_" followed by the randomized number. Overall, the generated EatxtProposalProvider encompasses 188 such methods.

Figure 7 depicts a screenshot of the content-assist proposal for a new FunctionFlowPort instance.

4.3 Formatters

Formatting text documents (e.g., Java source code) in the Eclipse IDE is the process of rearranging the documents' texts without semantically changing their contents. Xtext in principle supports this process as well, but does not provide out-of-the-box formatters (Eclipse Foundation, 2023e).

Instead, it requires that the language engineer extends an abstract formatter class and implements a corresponding dispatcher method for any metaclass to be formatted in its text representation. These methods are called when the formatting process is triggered and apply a series of text replacements for the corresponding model objects. As formatting should treat the text document and its particular contents in a uniform way, the implementation of these methods is highly repetitive.

In order to automate this tedious task and to provide a way of formatting the particular model elements uniformly, we implemented the generator fragment formatting2/EatxtFormatter2Fragment.xtend as part of the plugin org.bumble.eatxt (cf. fragment of the plugin in Figure 3). This fragment is executed by the MWE2 workflow and automatically generates the formatter class formatting2/EatxtFormatter.xtend as part of the same plugin. The fragment iterates over the metaclasses of the metamodel and generates the corresponding dispatcher method for each container metaclass. Furthermore, the fragment generates the individual formatting methods for all nested containments or sub-elements of these container classes calls. The generated formatter class in the EATXT case encompasses 51 dispatch methods and 141 calls of the formatting methods for the nested sub-elements.

4.4 Scoping for Cross-references

Programming languages, DSLs, and metamodels typically provide means to establish links between the semantic concepts defined in the corresponding models (e.g., for specifying the type of a language concept through referencing a different type concept). In the Xtext language development framework, such links are called *cross-references* (Eclipse Foundation, 2023d). The *scoping API* of Xtext provides the means for finding the target of a cross-reference based on its source context (Eclipse Foundation, 2023f). For example, the FunctionFlowPort instances in the lines 7–9 and 14–17 in Figure 2 are source contexts, and their elements type are cross-references pointing to target objects typed by the metaclass EADatatype (cf. Figure 1). If the target concept is nested in a container hierarchy (e.g., hierarchies of packages like the EAPackage instance DataTypes in lines 1–3 in Figure 2), this procedure for finding a cross-reference target particularly includes the computation of the scope within the nested container hierarchy.

Xtext provides a default out-of-the-box approach for the cross-reference scoping within container hierarchies by means of a *scope provider* (Eclipse Foundation, 2023f), where the scope provider also enables content-assist for the cross-reference targets. In this default approach, the fully qualified name of the cross-reference target in the container hierarchy is only proposed by the content-assist if the target is part of a different container. For example in Figure 2, the type cross-reference in line 9 as part of the container hierar-

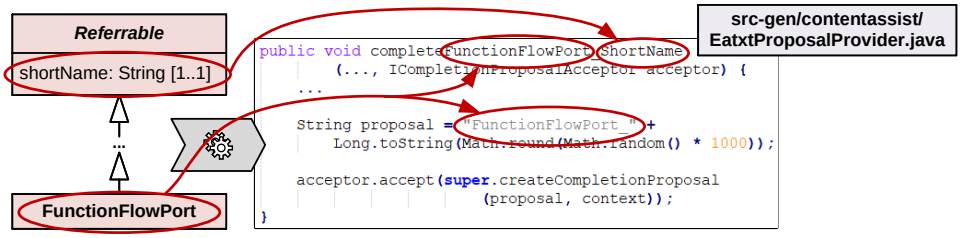

Figure 6: Exemplary Generation of a Content-assist Method for Unique Names of new FunctionFlowPort Model Elements

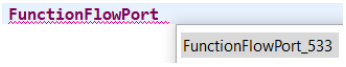

Figure 7: Content-assist for a new Model Element

chy FcnDesignTypes.WiperCtrl.WipingCmd points to the model element Integer_uint8 as part of the container DataTypes). In contrast, if the target is part of the same container as the source context, then only the plain name of the target is proposed but not its fully qualified name.

In the case of EATXT, a requirement is to provide content-assist that always proposes the fully qualified name. This requires a custom implementation of the scope provider (see also (Latifaj et al., 2021) for a different use case requiring such a custom implementation). In such a custom implementation, the DSL engineers have to compute for any source context metaclass the corresponding cross-reference target candidate metaclasses. For example, for the source context metaclass FunctionFlowPort and its cross-reference type, they have to realize that the target candidates are instances of the metaclass EADatatype (cf. Figure 1). In this context, multiple cross-reference target metaclasses are possible if the source context metaclass has multiple cross-references.

Basically, this computation is straightforward, but needs to consider all cross-references of the underlying metamodel, resulting in a large switch-case statement (i.e., if the source context of a cross-reference is an instance of a certain metaclass, return all candidates that are instances of the metaclasses of all target references). Particularly for large grammars and metamodels, this straightforward custom implementation is very awkward. Beyond that, both the Xtext default approach and custom implementation suffer from performance issues for large grammars and metamodels with many cross-references. In the runtime editor, the target candidate types are always computed on any content-assist keystroke for the given source context type. In the worst case, the lookup needs to traverse the entire switch-case-statement, which consists of all n possible cases for a complexity of $O(n)$.

As an automatic solution for this challenge in EATXT, we generate a cross-references lookup map in the activator of the plugin (cf. Activator.java in the plugin org.bumble.eatxt in Figure 3). This generation traverses the metamodel exactly once with a complexity of $O(n)$. This lookup map contains the corresponding type of the cross-reference target for any source context type, and we compute it by iterating over all cross-references in the metamodel. We generate the lookup map during the first activation of the plugin. After that, the scoping/EatxtScopeProvider.java accesses it via an interface but does not need to perform the same computation on every cross-reference content-assist keystroke. The lookup map is implemented as a Java HashMap whose get() method has a complexity of $O(1)$ in most cases. In the EATXT case, the map encompasses the source context metaclasses and corresponding target metaclasses for 261 cross-references of the EAST-ADL metamodel. Figure 8 depicts a screenshot of the content-assist for cross-referencing an existing model element in a different container hierarchy.

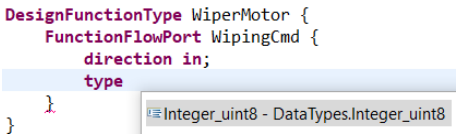

Figure 8: Cross-referencing an Existing Model Element

The solution has two advantages, in particular for large grammars and metamodels. First, we automate the awkward custom implementation of a scope provider with a generic solution that exploits the underlying metamodel cross-references structure. Second, it significantly improves the performance of the Xtext scope provider approach by computing a cross-reference lookup table once at plugin activation, instead of computing the target types on every content-assist keystroke for a source context. Thus, the expensive operation takes place only once rather than repeatedly at runtime.

5 Conclusion and Outlook

Our solutions to the four challenges of template proposals, formatters, content-assist, and scoping have been developed and evaluated in the context of EATXT, but are not only useful there. Instead, we argue that similar approaches are helpful for any highly structured abstract syntax for which Xtext is used to generate editors for a textual concrete syntax. A specific language will benefit from adaptations, especially in the area of scoping. As such, our solutions are not generic enough to contribute to Xtext directly, but will hopefully serve as a blueprint that is useful to other engineers with similar challenges.

We view our contributions in the light of a future language workbench that supports the blended modeling for large DSLs that evolve. As described in our previous work (Holtmann et al., 2022), EAST-ADL is such a language and support for its evolution within Eclipse is a relatively new capability. Our work further supports this approach by providing a streamlined way to generate the editors whenever the language changes. When coupled with the generation of a graphical editor (see, e.g., (Cooper and Kolovos, 2019)), this brings us closer to the ideal of a *blended modeling language workbench*.

ACKNOWLEDGEMENTS

Parts of this research were sponsored by Vinnova under grant agreement nr. 2019-02382 as part of the ITEA4 project BUMBLE.

REFERENCES

Ciccozzi, F., Tichy, M., Vangheluwe, H., and Weyns, D. (2019). Blended Modelling – What, Why and How. In *22nd ACM/IEEE Intl. Conf. on Model Driven Engineering Languages and Systems (MODELS) Companion Proceedings*, pages 425–430. IEEE.

Cooper, J. and Kolovos, D. (2019). Engineering hybrid graphical-textual languages with sirius and xtext: Requirements and challenges. In *2019 ACM/IEEE 22nd Intl. Conf. on Model Driven Engineering Languages and Systems (MODELS) Companion Proceedings*, pages 322–325. IEEE.

Cuenot, P., Chen, D., Gerard, S., Lönn, H., Reiser, M.-O., Servat, D., Sjostedt, C.-J., Kolagari, R. T., Torngren, M., and Weber, M. (2007). Managing complexity of automotive electronics using the East-ADL. In *12th IEEE Intl. Conf. on Engineering Complex Computer Systems (ICECCS 2007)*, pages 353–358. IEEE.

David, I., Latifaj, M., Pietron, J., Zhang, W., Ciccozzi, F., Malavolta, I., Raschke, A., Steghöfer, J.-P., and Hebig, R. (2022). Blended modeling in commercial and open-source model-driven software engineering tools: A systematic study. *Software & Systems Modeling*.

Debruyne, V., Simonot-Lion, F., and Trinquet, Y. (2004). EAST-ADL—an architecture description language. In *IFIP World Computer Congress, TC 2*, pages 181–195. Springer.

Eclipse Foundation (2023a). Xtend. https://www.eclipse.org/xtend/. Last accessed Jan 2023.

Eclipse Foundation (2023b). Xtext reference documentation: Configuration. https://www.eclipse.org/Xtext/documentation/302_configuration.html. Last accessed Jan 2023.

Eclipse Foundation (2023c). Xtext reference documentation: Content assist. https://www.eclipse.org/Xtext/documentation/310_eclipse_support.html#content-assist. Last accessed Jan 2023.

Eclipse Foundation (2023d). Xtext reference documentation: Cross-references. https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html#cross-references. Last accessed Jan 2023.

Eclipse Foundation (2023e). Xtext reference documentation: Formatting. https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#formatting. Last accessed Jan 2023.

Eclipse Foundation (2023f). Xtext reference documentation: Scoping. https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#scoping. Last accessed Jan 2023.

Eclipse Foundation (2023g). Xtext reference documentation: Template proposals. https://www.eclipse.org/Xtext/documentation/310_eclipse_support.html#templates. Last accessed Jan 2023.

Eysholdt, M. and Behrens, H. (2010). Xtext: implement your language faster than the quick and dirty way. In *ACM Intl. Conf. on Object oriented programming systems languages and applications companion*, pages 307–309.

Holtmann, J., Steghöfer, J., and Lönn, H. (2022). Migrating from proprietary tools to open-source software for EAST-ADL metamodel generation and evolution. In Kühn, T. and Sousa, V., editors, *25th Intl. Conf. on Model Driven Engineering Languages and Systems (MODELS) Companion Proceedings*, pages 7–11. ACM.

Holtmann, J., Steghöfer, J.-P., and Zhang, W. (2023). EATXT implementation excerpt. https://github.com/joerg-holtmann/EATXT4MODELSWARD23. Last accessed Jan 2023.

Latifaj, M., Ciccozzi, F., Mohlin, M., and Posse, E. (2021). Towards Automated Support for Blended Modelling of UML-RT Embedded Software Architectures. In *ECSA (Companion)*.

Neubauer, P., Bergmayr, A., Mayerhofer, T., Troya, J., and Wimmer, M. (2015). XMLText: From XML schema to xtext. In *2015 ACM SIGPLAN Intl. Conf. on Software Language Engineering*, pages 71–76.