Proceedings Paper:
Predoaia, Ionut orcid.org/0000-0002-2009-4054, Kolovos, Dimitris orcid.org/0000-0002-1724-6563 and Garcia-Dominguez, Antonio orcid.org/0000-0002-4744-9150 (2024) Hybrid Graphical-Textual DSL Editors: Vision, Requirements and Challenges. In: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems Companion, MODELS-C 2024. 2024 ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, 22-27 Sep 2024 MODELS Companion '24 . ACM , AUT , pp. 1156-1160.

https://doi.org/10.1145/3652620.3688346

Hybrid Graphical-Textual DSL Editors: Vision, Requirements and Challenges

Ionut Predoaia
University of York
York, United Kingdom
ionut.predoaia@york.ac.uk

Dimitris Kolovos
University of York
York, United Kingdom
dimitris.kolovos@york.ac.uk

Antonio García-Domínguez
University of York
York, United Kingdom
a.garcia-dominguez@york.ac.uk

ABSTRACT

Hybrid graphical-textual domain-specific languages can deliver the best of both worlds of graphical and textual modelling: an intuitive graphical syntax for some parts of the language and a concise textual syntax for others. This paper discusses the requirements of hybrid graphical-textual domain-specific languages and their supporting editors, highlighting challenges and potential enhancements. We present our vision for future developments, aiming to simplify the process of engineering such languages and additionally, to expand their capabilities.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Model-driven software engineering**; • **Theory of computation → Grammars and context-free languages**.

KEYWORDS

Hybrid Notations, Graphical-Textual Modelling, Code Generation, Static Analysis, Grammar, Xtext, Sirius

ACM Reference Format:
Ionut Predoaia, Dimitris Kolovos, and Antonio García-Domínguez. 2024. Hybrid Graphical-Textual DSL Editors: Vision, Requirements and Challenges. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3652620.3688346

1 INTRODUCTION

Hybrid graphical-textual domain-specific languages (DSLs) are languages that have a hybrid notation. They comprise a part-graphical and part-textual syntax, where the graphical part is commonly used for the representation of high-level domain concepts and the textual part is used for capturing complex expressions and behaviour [9, 10]. In essence, they are graphical DSLs containing embedded textual expressions, where the graphical and textual syntaxes mostly cover mutually exclusive parts of the abstract syntax, although overlaps can exist. Hybrid graphical-textual DSLs can deliver the best of both worlds of graphical and textual modelling [4], due to the synergies obtained when combining both notations.

For brevity, the term *hybrid* will be used instead of the term *hybrid graphical-textual*. Hybrid DSLs are effectively used through hybrid model editors, alternatively called hybrid DSL editors, as they enable editing a single domain model through graphical and textual notations.

This paper presents the required capabilities of hybrid DSLs and their supporting editors, as argued in the literature [4, 9, 10], highlighting challenges and potential improvements. We present our vision for future developments, intending to simplify the process of engineering hybrid DSL editors. We propose techniques such as the automatic generation of grammars, compliance validation of grammars, grammar inspection of referenced types, and an approach for post-processing derived model elements.

Sections 2 and 3 illustrate a running example and the requirements of hybrid DSL editors. Sections 4 and 5 present Graphite, a tool for engineering hybrid DSL editors, and a vision for its development. Section 6 presents related work. Finally, Section 7 concludes the paper, providing future work directions.

2 RUNNING EXAMPLE

This section presents a minimal contrived example that will be used to showcase requirements, challenges and our vision of hybrid DSL editors. Listing 1 presents the metamodel of a DSL for modelling project plans, that has been defined in Emfatic [5], a textual syntax for Ecore metamodels. The metamodel of the *Project Workloads DSL* specifies that the root of the domain is a *Project* containing lists of *tasks* and *people*. A *Task* has a *name* and a list of *efforts*, where each *Effort* is assigned to a *person* and has a number of *months*.

For the purpose of this example, we will assume that stakeholders prefer a hybrid syntax for the DSL, where *tasks* and *people* are modelled graphically, but the allocation of *efforts* is captured using an embedded textual notation as shown in Figure 1. Figure 1 illustrates a *Project* in a hybrid DSL editor, containing *tasks* and *people* modelled graphically, and *efforts* that are modelled through a YAML-like textual syntax. The edges mark the *dependencies* and *leader* of each task. The *efforts* are defined on separate lines as key-value pairs having the form {*person*}:{*months*}. Accordingly, the model elements of type *Task* and *Person* represent the graphical parts of the model, whereas those of type *Effort* represent the textual parts of the model. The *task* named *Implementation* is selected in the diagram, therefore its properties, i.e., *name* and *efforts*, are displayed in the properties view. Each line from the *efforts* textual expression represents an *effort* model element, e.g., the second line is an *effort* that refers to the *person* named *Bob* and has a value of 6 *months*.

MODELS Companion '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3688346

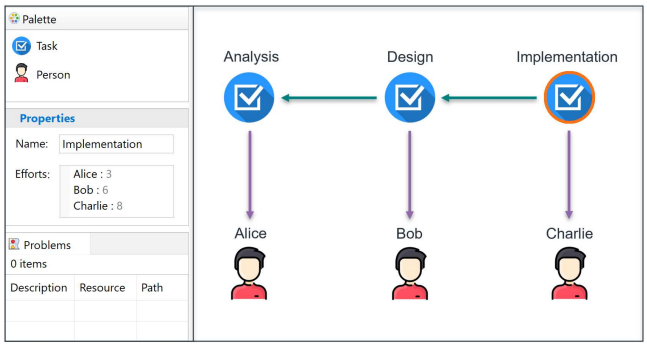

Figure 1: Hybrid DSL Editor [10]

```
@namespace(uri="ProjectWorkloadsDSL")
package workload;

class Project {
   val Task[*] tasks;
   val Person[*] people;
}

class Task {
   attr String name;
   val Effort[*] efforts;
   ref Person leader;
   ref Task[*] dependencies;
}

class Person {
   attr String name;
}

class Effort {
   ref Person person;
   attr int months;
}
```

Listing 1: Metamodel of the *Project Workloads DSL*

3 REQUIREMENTS

Hybrid DSL editors should ideally provide the following capabilities, as argued in the literature [4, 9, 10].

3.1 Smart Textual Editors

Most graphical model editors already support plain textual editors for editing the textual properties of model elements. When modelling the properties expressed through a textual syntax (e.g., the *efforts*), it would be ideal to benefit from the developer assistance features that are typically included in most of the modern integrated development environments (IDEs). Therefore, it is desirable to model the *efforts* through smart textual editors supporting syntax-aware editing features, such as syntax highlighting, auto-completion and error detection [4, 9, 10]. For instance, as a user types "*Ali*" in the smart textual editor used for defining the *efforts*, an auto-completion menu should be displayed to show all *people* from the diagram whose *name* starts with "*Ali*", i.e., *Alice*.

3.2 Textual-Graphical Cross-Referencing

To be able to define complex expressions and behaviour, the textual expressions must be able to reference graphical model elements that have been defined in diagrams [4, 9, 10]. For instance, the *efforts* textual expression from Figure 1 references the *people* defined in the diagram, i.e., *Alice*, *Bob* and *Charlie*. A desirable feature is to support navigation from textual expressions to referenced model elements from diagrams. For example, an action such as control-click performed in the smart textual editor on *Charlie* should trigger the navigation to the diagram definition of the *person* named *Charlie*.

3.3 Consistency Enforcement

To avoid potential inconsistencies in the model, it is desirable to have consistency between the graphical and textual parts of the model automatically enforced by the hybrid DSL editor [9, 10]. To this end, when a graphical model element from the diagram is renamed or deleted, all textual expressions that were previously referencing the respective model element must be updated accordingly. For instance, when in the diagram, the *person* named *Bob* is renamed to *Robert*, then in the smart textual editor, the second line from the textual expression should be updated accordingly by replacing *Bob* with *Robert*. Furthermore, the hybrid DSL editor should tolerate temporary inconsistencies, to be able to save models when textual expressions are not in a consistent state with the rest of the model, e.g., when reference resolution fails.

3.4 Uniform Error Reporting

The hybrid DSL editor should uniformly report errors related to the textual and graphical parts of the model, to inform users about any existing inconsistencies [4, 9, 10]. Furthermore, it would be desirable to support navigation to problematic model elements from a reported error, to avoid the overhead of locating them manually. For example, the textual expression from Figure 1 references the *person* named *Charlie* from the diagram, however, if this *person* did not exist in the diagram, this should trigger the hybrid DSL editor to report the inconsistency as an error.

3.5 Integrated Abstract Syntax Graph

For the purpose of performing model management over the entire model, it is required to expose the model as a single unified abstract syntax graph (ASG) that integrates elements from both its textual and graphical parts [4, 9, 10]. For instance, the underlying semantic model from Figure 1 must be exposed to model management programs as a unified ASG that integrates the textual parts of the model, i.e., the *efforts*, and the graphical parts, i.e., the *tasks* and the *people*. Consequently, the *efforts* textual expression must not be exposed to model management programs as plain text, but rather as a list of *effort* model elements that can be accessed, queried and manipulated as part of a model management operation.

4 GRAPHITE

Our work from [10] presents techniques for addressing the requirements from Section 3. The techniques described in [10] have been implemented in a tool named Graphite[1], which streamlines the development of hybrid DSL editors by using model transformations. For using the approach described in [10], one must define

[1] https://github.com/epsilonlabs/graphite

```
class Task {
   ...
   @syntax(grammar="gEfforts", derive="efforts")
   attr String effortsExpression;
   val Effort[*] efforts;
   ...
}
```

Listing 2: Annotated Metamodel

```
1  grammar gEfforts
2     with org.eclipse.xtext.common.Terminals
3
4  import "ProjectWorkloadsDSL"
5  import "http://www.eclipse.org/emf/2002/Ecore"
6
7  Main returns Task:
8     {Task}
9     (efforts+=Effort (NEWLINE efforts+=Effort)*)?;
10
11 Effort returns Effort:
12    {Effort}
13    (person=[Person])? ':' months=INT;
14
15 terminal NEWLINE:
16    ('␣'|'\t')* '\r'? '\n' ('␣'|'\t')*;
```

Listing 3: *Efforts* Grammar of a YAML-like Textual Syntax

the graphical syntax of the DSL through a Sirius Viewpoint Specification Model, and the textual syntaxes through Xtext grammars. In addition, the metamodel of the DSL must be modified, by adding a string attribute and an annotation for each property from the metamodel that one would like to express through a textual syntax.

Listing 2 shows how the *Task* metaclass must be modified to express the *efforts* through a textual syntax. The *effortsExpression* string attribute has been added to store the textual representation of the *efforts* list. Additionally, an annotation has been added to define the mapping between the added string attribute (i.e., *effortsExpression*) and the underlying model elements (i.e., *efforts*). The annotation specifies the grammar that is used for parsing the *efforts* textual expression, and the property in which the derived model elements are stored, i.e., when parsing *effortsExpression* with the *gEfforts* grammar, the derived model elements are assigned to the *efforts* property. The properties expressed through a textual syntax and their textual representation remain synchronised, i.e., the *efforts* and *effortsExpression* are bidirectionally synchronised.

The *gEfforts* grammar from Listing 3 defines the YAML-like textual syntax used for modelling the *efforts*. The grammar specifies that whenever the textual representation of the *efforts* is parsed, a *Task* that contains a list of *Effort* model elements is derived. Therefore, when using this approach, grammars must adhere to a structure that is compatible with the annotated metamodel, as the root model element that is derived must be an instance of the metaclass (e.g., *Task*) that contains the property being expressed through a textual syntax (e.g., *efforts*). By following the described approach, one can finally execute a model-to-text transformation that takes as input the annotated metamodel for automatically generating code that configures hybrid DSL editors that meet all requirements from Section 3.

5 VISION FOR FUTURE DEVELOPMENTS

This section describes our vision for improving Graphite. We propose a set of techniques that aim to further simplify the process of engineering hybrid DSL editors, and additionally, enhance their capabilities.

5.1 Automatic Generation of Grammars

As described in the last paragraph of Section 4, a grammar that is used for specifying an embedded textual syntax must adhere to a structure that is compatible with the annotated metamodel. Specifically, the root model element that is derived when parsing a textual expression using the grammar must be an instance of the container metaclass (i.e., *Task* in our example) of the property being expressed with a textual syntax (e.g., *efforts*). Furthermore, only the property being expressed through a textual syntax must be populated by the grammar, e.g., only the *efforts* property of a *Task* must be populated, as the other properties of a *Task*, i.e., *name* and *effortsExpression*, do not need to be set by the grammar.

For creating an Xtext grammar, a new *Xtext project from Ecore* must be created, as an existing Ecore metamodel must be imported and referenced throughout the grammar. Before creating the project, a user must first select the entry rule that specifies the metaclass of the derived root model element. Next, Xtext automatically generates a skeleton of the grammar, containing all grammar rules required to populate the root model element and all its child model elements recursively.

One could leverage the capability of Xtext to automatically generate a grammar for the purpose of generating all grammars specified in the annotated metamodel. This would simplify the language engineering experience, by having the skeleton of a compliant grammar as a starting point, which can be customised further at a later point. To this end, the Xtext API which automatically generates a grammar must be customised to generate only grammar rules for populating the property being expressed through a textual syntax (e.g., the *efforts*), and ignore all other properties. Then, a model management program can be executed to statically analyse the annotated metamodel, and for each annotation that defines the mapping to a grammar, the customised Xtext API is called to automatically generate a grammar with an entry rule that derives a model element that is an instance of the container metaclass. For instance, with the annotated metamodel from Listing 2, one grammar would be automatically generated for specifying the *efforts* list, which would contain an entry rule that derives a model element of type *Task*, containing only the *efforts* list (i.e., the grammar rule at lines 7–9 from Listing 3), as the other properties, i.e, *name* and *effortsExpression* remain unset. Furthermore, the generated grammar would contain a rule for deriving model elements of type *Effort* (i.e., the grammar rule at lines 11–13 from Listing 3), and additional rules that recursively populate all child elements of an *Effort*, if any.

5.2 Compliance Validation of Grammars

Graphite can only be applied with a compliant grammar, as described previously. Additionally, a compliant grammar must also populate all non-transient properties of a derived model element, except for the root (i.e., *Task*). For instance, the grammar rule (lines 11–13 from Listing 3) that derives an *Effort* model element must set

```
class PostProcessor implements IPostProcessor {
   @Override
   public Object transform(EObject object) {
      ...
   }
}
```

Listing 4: Post-Processor Service

```
@syntax(grammar=..., postprocess="PostProcessor")
attr String effortsExpression;
val Effort[*] efforts;
```

Listing 5: Annotation Attribute for Post-Processor

all its properties, i.e., *person* and *months*. Otherwise, if not all properties are set, data would be lost when carrying out bidirectional synchronisation between the *efforts* and *effortsExpression* through serialisation and deserialisation. Furthermore, this also recursively applies to all child model elements of a derived model element, i.e., if an *Effort* contained any child elements, then all their properties should have also been set. Therefore, it would be useful to validate whether a grammar is compliant. For this purpose, a model validation operation can be used, which confirms the compliance of a grammar, or informs the language engineer about any changes that must be made to the grammar to make it compliant.

The model validation operation should take as input two models, the annotated metamodel and the Xtext grammar, considering that both are EMF models at their core. The operation first identifies the metaclass which contains an annotation for the grammar being validated. For instance, when passing the annotated metamodel from Listing 2 and the grammar from Listing 3 as input to the model validation operation, the metamodel would be statically analysed, by searching for the metaclass which contains an annotation associated with the grammar named *gEfforts*. All metaclasses are iterated until the *Task* metaclass is identified as being the one containing the annotation associated with the grammar. Then, the grammar is statically analysed, by checking whether it contains a grammar rule returning a model element of type *Task*, such as the one at lines 7–9 from Listing 3. The grammar rule must only set the property specified in the *derive* attribute of the annotation, i.e., the *efforts*. Furthermore, it must be checked whether a grammar rule exists for instantiating *Effort* model elements, and whether additional rules exist for recursively setting their child elements, if any. The grammar rules instantiating *Effort* model elements and their child model elements, are checked to verify whether they set the value of all their non-transient properties, by taking into account the structure of the metamodel.

5.3 Grammar Inspection of Referenced Types

Graphite has a limitation in the case of unresolved references. The way in which consistency is automatically enforced is by attaching event listeners to referenced model elements whenever a textual expression is parsed. For example, when the textual expression from Figure 1 is parsed, the referenced model elements of type *Person* are identified, i.e., the persons named *Alice*, *Bob* and *Charlie*, and an event listener is attached to each. When the *name* of the *person Alice* is changed, then the *efforts* list is serialised and the resulting string overwrites the textual expression. However, this technique does not work in the case of an unresolved reference. For instance, if the *person* named *Charlie* did not exist in the diagram, reference resolution would fail when parsing the textual expression from Figure 1. In this scenario, if another *person* named *David*, which is not shown in the diagram, is renamed into *Charlie*, then reference resolution should ideally be triggered to resolve the prior issue, and then to attach an event listener to the *person* named *Charlie*. However, in the current solution, reference resolution is not triggered in such a case. Nevertheless, an error would be reported, therefore the user is made aware of how to solve the issue.

To address the mentioned issue that occurs in the case of an unresolved reference, a naive approach would be to trigger reference resolution for the *efforts* list whenever any property of any model element has changed. An efficient technique would be to statically analyse the grammar of the *efforts*, to identify the types that are referenced in a textual expression. For instance, the grammar rule at lines 11–13 from Listing 3 defines a reference to model elements of type *Person*, at line 13. By using a model transformation that takes as input the grammar, we could identify the referenced types of model elements from the grammar rules, to automatically generate code for setting event listeners tailored to *Person* model elements. Therefore, when the *name* of a model element of type *Person* changes, this would trigger a reference resolution operation for the *efforts* list. In our running example, there is only one property expressed through a textual syntax, and one grammar is used, however, multiple grammars could exist that are associated with numerous properties. With this technique, mappings must be defined between the grammars and referenced types, to know for which properties of the metamodel to trigger reference resolution.

5.4 Post-Processing of Derived Model Elements

One could benefit from applying a post-processing operation over the derived model elements, to leverage complex business logic to transform the model elements into another form, that could have not been carried out through the grammar's parser. For instance, when parsing *effortsExpression*, we might wish to remove all derived *effort* model elements that have a value of 0 *months*. To this end, one could define a post-processor service as the one from Listing 4, and then store a reference to the post-processor service in an attribute (i.e., *postprocess*) of the annotation, as in Listing 5. Then, the model-to-text transformation from Graphite which automatically generates code for configuring hybrid DSL editors [10], would have to be modified such that it takes into account the *postprocess* annotation attribute from Listing 5, to apply the post-processor service over the derived model elements.

6 RELATED WORK

Hybrid DSLs. In addition to Graphite, hybrid DSLs could alternatively be engineered using projectional editors such as JetBrains MPS [6]. The works from [1, 3, 8, 12] present techniques for engineering hybrid DSLs and their supporting editors, however, unlike our work, they heavily rely on hand-written code and have limitations regarding consistency enforcement and error reporting.

Other related works are based on blended modelling [2], which focuses on providing several graphical and textual notations for the same concepts of the abstract syntax, while keeping the different notations synchronised. However, hybrid DSLs are concerned with using graphical and textual notations for different concepts of the abstract syntax, while maintaining the consistency of the references between the graphical and textual parts.

Grammar Inspection and Generation. The work from [11] proposes a technique based on model transformations and static analysis to transform an Xtext grammar into a Java library for SonarQube, used for evaluating the quality of programs written with Xtext-designed DSLs. An approach involving grammar transformations is presented in [13], which loads a default Xtext-generated grammar into memory as a model, and then applies transformation rules on it to transform it into an expert grammar. An alternative to the grammar post-processing technique from [13] is to directly customise the grammar generator of Xtext, as presented in [7], similarly to how we described in our vision. Other related works concern the co-evolution between metamodels and generated grammars [13, 14].

7 CONCLUSIONS AND FUTURE WORK

This paper presented a set of requirements for hybrid DSLs and their supporting editors, highlighting what can be achieved at present and what challenges remain. We described our vision for future developments, proposing techniques such as the automatic generation of grammars, compliance validation of grammars, grammar inspection of referenced types, and finally, post-processing of derived model elements. In future work, we plan to investigate unified searching capabilities across the textual and graphical parts of the model.

ACKNOWLEDGMENT

The work in this paper has been funded through NetApp, the HICLASS InnovateUK project (contract no. 113213), and the SCHEME InnovateUK project (contract no. 10065634).

REFERENCES

[1] Altran. 2022. *Xtext Sirius integration.* [Online]. Available: https://altran-mde.github.io/xtext-sirius-integration.io, (Last Accessed: 2024-08-15).
[2] Federico Ciccozzi, Matthias Tichy, Hans Vangheluwe, and Danny Weyns. 2019. Blended Modelling - What, Why and How. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 425–430.
[3] Justin Cooper. 2018. *A Framework to Embed Textual Domain Specific Languages in Graphical Model Editors.* Master's thesis. University of York.
[4] Justin Cooper and Dimitris Kolovos. 2019. Engineering Hybrid Graphical-Textual Languages with Sirius and Xtext: Requirements and Challenges. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 322–325.
[5] Eclipse. 2024. *Emfatic.* [Online]. Available: https://eclipse.org/emfatic, (Last Accessed: 2024-08-15).
[6] JetBrains. 2024. *JetBrains MPS Website.* [Online]. Available: https://www.jetbrains.com/mps, (Last Accessed: 2024-08-15).
[7] Patrick Neubauer, Alexander Bergmayr, Tanja Mayerhofer, Javier Troya, and Manuel Wimmer. 2015. XMLText: from XML schema to Xtext. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*. 71–76.
[8] Obeo and TypeFox. 2017. *Xtext Sirius integration - white paper.* [Online]. Available: https://www.obeodesigner.com/resource/white-paper/WhitePaper_XtextSirius_EN.pdf, (Last Accessed: 2024-08-15).
[9] Ionut Predoaia. 2023. Towards Systematic Engineering of Hybrid Graphical-Textual Domain-Specific Languages. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 153–158.
[10] Ionut Predoaia, Dimitris Kolovos, Matthias Lenk, and Antonio García-Domínguez. 2023. Streamlining the Development of Hybrid Graphical-Textual Model Editors for Domain-Specific Languages. *Journal of Object Technology* 22, 2 (2023).
[11] Iván Ruiz Rube, Tatiana Person, and Juan Manuel Dodero. 2020. Static analysis of textual models. *XXI Jornadas de Ingeniería del Software y Bases de Datos* 219 (2020), 269.
[12] Markus Scheidgen. 2008. Textual Modelling Embedded into Graphical Modelling. In *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 153–168.
[13] Weixing Zhang, Jörg Holtmann, Daniel Strüber, Regina Hebig, and Jan-Philipp Steghöfer. 2024. Supporting meta-model-based language evolution and rapid prototyping with automated grammar transformation. *Journal of Systems and Software* 214 (2024), 112069.
[14] Weixing Zhang, Jan-Philipp Steghöfer, Regina Hebig, and Daniel Strüber. 2023. A Rapid Prototyping Language Workbench for Textual DSLs based on Xtext: Vision and Progress. *arXiv preprint arXiv:2309.04347* (2023).