
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 (2025) Graphite: Automated Development of Hybrid Graphical-Textual DSL Editors. In: Proceedings of the ACM / IEEE 28th International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). 28th International Conference on Model Driven Engineering Languages and Systems (MODELS), 05-10 Oct 2025 , USA.

Graphite: Automated Development of Hybrid Graphical-Textual DSL Editors

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

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

Antonio García-Domínguez
University of 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, by providing a graphical syntax for some parts of the language and a textual syntax for others. Graphite is a tool that facilitates the automated development of hybrid graphical-textual model editors for domain-specific languages. This paper outlines the capabilities of the hybrid editors generated by Graphite: smart textual editors, textual-graphical cross-referencing, integrated refactoring, consistency enforcement, tolerance of temporary inconsistencies, integrated abstract syntax graph, uniform error reporting, and conditional storage of derived model elements. Furthermore, the language engineering process employed by Graphite is demonstrated.**

***Index Terms*—Hybrid DSL Editor, Code Generation, EMF, Graphical-Textual Modelling, Grammar, Graphite, Xtext, Sirius**

I. Introduction

Hybrid graphical-textual domain-specific languages (DSLs) are languages with a hybrid concrete syntax. The graphical part of the hybrid syntax is commonly used for representing high-level domain concepts, whereas the textual part typically captures complex expressions and behaviour. In essence, they are graphical DSLs containing embedded textual expressions. Accordingly, hybrid graphical-textual DSLs can deliver the best of both worlds of graphical and textual modelling, as textual representations can reduce the number of clicks when creating and editing models, whereas graphical representations can reduce the time spent linking model elements together [1].

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. When using a hybrid DSL, some parts of the model are graphical, i.e., they are expressed with a graphical syntax, whereas others are textual, i.e., they are expressed with a textual syntax. The term *graphical model elements* is used to refer to the graphical parts of the model, whereas the term *textual model elements* refers to textual parts of the model that are expressed through textual expressions.

This paper presents Graphite[1], an open-source tool that facilitates the automated development of hybrid DSL editors. The work in this paper extends our prior works [2]–[5] by packaging Graphite into an installable tool with a user interface, and by providing additional facilities such as metamodel and grammar validation, automatic generation of grammars and integrated refactoring. Hybrid DSL editors have been developed using Graphite for the following DSLs: the Project Scheduling DSL [3], the NetApp Cloud Services DSL [2], the Fuzzing-based Testing DSL [6], and the Structurizr DSL [5].

The generated hybrid DSL editors are EMF-based, and rely on the integration of the Sirius graphical modelling framework and the Xtext textual modelling framework. For generating a hybrid DSL editor, Graphite requires as input an Ecore metamodel, an EMF generator model, a Sirius Viewpoint Specification Model (VSM) [7], and one or more Xtext grammars. The generated hybrid DSL editors provide the following key features: smart textual editors, textual-graphical cross-referencing, tolerance of temporary inconsistencies, uniform error reporting, consistency enforcement and refactoring.

Graphite's language engineering process is demonstrated in this paper. The process can be outlined as follows: the language engineer (1) defines the metamodel and then extends it with string attributes and annotations, (2) changes a property of the generator model and then launches the EMF code generator, (3) defines the graphical and textual syntaxes using Sirius and Xtext, and (4) executes the code generator of Graphite to produce a hybrid DSL editor.

Paper structure. Section II presents related work. Section III describes a running example. Section IV outlines the capabilities of Graphite. Section V demonstrates Graphite's language engineering process. Finally, Section VI concludes the paper and provides directions for future work.

II. Related Work

In addition to Graphite, hybrid DSL editors could alternatively be engineered using projectional editors such as JetBrains MPS [8]. The works from [9]–[12] present techniques for engineering hybrid DSLs and their supporting editors, however, unlike our work, they heavily rely on hand-written code. Other related works are based on blended modelling [13], which focuses on providing several graphical and textual notations for the same concepts of the abstract syntax, while keeping the different notations synchronised. However, in our work we are concerned with DSLs that use graphical and textual notations for different concepts of the abstract syntax,

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

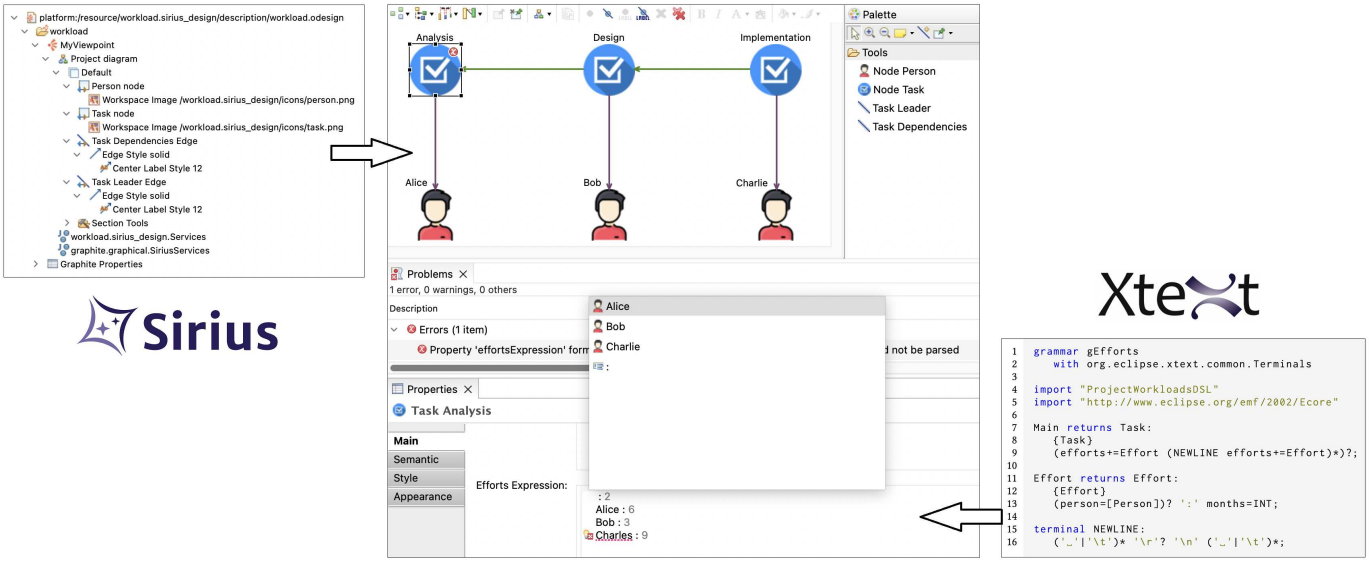

Fig. 1. Hybrid DSL Editor

while maintaining the consistency of the references between the graphical and textual parts.

None of the prior works propose techniques and facilities for integrated refactoring, consistency enforcement, tolerance of temporary inconsistencies and conditional storage of derived model elements. Furthermore, in contrast with our automated approach, the prior solutions are realised via a considerable amount of hand-written code. Moreover, a declarative specification is used in prior works for specifying which parts of the language's syntax are graphical; however, no techniques have been proposed for declaratively defining the textual parts of the syntax.

III. Running Example

This section presents a minimal contrived example that will be used to showcase the capabilities of Graphite. Listings 1 and 2 present the metamodel (initial and after extension) of a DSL for modelling project plans that has been defined in Emfatic [14], a textual syntax for Ecore metamodels. The metamodel of the *Project Scheduling 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*.

We assume for the purpose of this example that stakeholders prefer a hybrid syntax for the DSL, where *tasks* and *people* are modelled graphically, but the *efforts* are specified using an embedded textual notation as shown in Figure 1. Figure 1 illustrates a hybrid DSL editor operating over a model conforming to the *Project Scheduling DSL*'s metamodel. The *tasks* and *people* are modelled graphically, and the *efforts* are modelled through a dictionary-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. Note that the graphical part of the syntax is specified using a Sirius VSM, whereas the textual part is defined using an Xtext grammar, as shown in Figure 1.

As the *task* named *Analysis* is selected in the diagram, 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. For example, the second line is an *effort* that refers to the *person* named *Alice* and has a value of 6 *months*.

IV. Graphite

Hybrid DSL editors developed with Graphite are composed of Sirius graphical diagram(s), Xtext textual editor(s), a Properties view for viewing and editing the properties of a selected model element, a Palette view containing various symbols that can be dragged and dropped on top of diagrams to create various types of model elements, and a Problems view that consistently reports errors in the model. In the following, we describe the capabilities provided by Graphite.

A. Tolerance of Temporary Inconsistencies

To be able to tolerate temporary inconsistencies in the hybrid DSL editor, Graphite requires the language engineer to modify the metamodel by adding a string attribute for each property that they desire to express through a textual syntax. This decision and alternatives considered have been discussed in detail in [2]. Therefore, one string attribute (i.e., `attr String effortsExpression`) has been added to the *Task* meta-class in the metamodel (see Listing 1), to store the *efforts* in a textual format. The *effortsExpression* string attribute (i.e., the textual expression) represents a textual representation of the *efforts* list (i.e., the textual model elements). Note that the content of the embedded textual editor is the value of *effortsExpression*. As shown in Listing 1, an annotation must be added to the metamodel to define a mapping between the added string attribute (i.e.,

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

Listing 1. Annotated Metamodel obtained by extending the initial metamodel with a string attribute and an annotation

```
@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 2. Initial Metamodel of the *Project Scheduling DSL*

effortsExpression), the property that it represents (i.e., *efforts*), and the grammar that is used for parsing, along with its entry rule. The *effortsExpression* is parsed according to the grammar and the derived model elements resulting from the parsing operation are then assigned to the *efforts* list. Note that the extended metamodel containing the added string attribute and annotation will be referred to as the *annotated metamodel*.

B. Integrated Abstract Syntax Graph

To be able to perform model management operations over the entire model such as model-to-model and model-to-text transformation, Graphite exposes the model over which the hybrid DSL editor operates as a single unified abstract syntax graph (ASG) that integrates elements from both its textual and graphical parts. For instance, the underlying semantic model from Figure 1 can 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 textual model elements (i.e., the *efforts* list) are not exposed to model management programs only as plain text, but also as a list of model elements of type *Effort* that can be accessed, queried and manipulated as part of a model management operation. For instance, one could write a script using the Epsilon Object Language (EOL) [15] to execute a *for loop* to iterate over the list of four *efforts* from Figure 1 to then print the number of *person*s having an *effort* higher than 6 *months*. It is worth noting that for ensuring an integrated ASG, bidirectional synchronisation is carried out between textual model elements (i.e., the *efforts* list) and their textual expression (i.e., the *effortsExpression* string attribute), to ensure the integrity of the ASG.

C. Smart Textual Editors

In the Properties view, the embedded textual editor used for editing the *effortsExpression* is a smart textual editor. The smart textual editor provides syntax-aware editing features, such as syntax highlighting, auto-completion, and error detection. In Figure 1, the user is editing the *effort* on the first line: while they are typing the *Person* assigned to the *effort*, an auto-completion menu is displayed, showing all *people* in the diagram. In addition, the *effort* on the fourth line references the *Person* named *Charles*, who does not exist in the diagram, therefore, an error marker is displayed on the left side of the smart textual editor to inform the user about the issue.

D. Textual-Graphical Cross-Referencing

To be able to define complex expressions and behaviour, the textual expressions written in the smart textual editors can reference graphical model elements that are defined in the diagram. For instance, the *efforts* textual expression references the *people* defined in the diagram, i.e., *Alice* and *Bob* (*Charles* does not exist). The hybrid DSL editor provides navigation from textual expressions to referenced model elements in diagrams. Performing *control-click* in the smart textual editor on *Alice* will trigger the navigation (selection) to the diagram definition of the *person* named *Alice*.

E. Integrated Refactoring

The hybrid DSL editor provides integrated refactoring through the smart textual editor. In the smart textual editor, one can *right-click* a referenced graphical model element (e.g., the *person* named *Bob*) in the textual expression and then a context menu is displayed, allowing the user to perform a refactoring operation over the selected model element. Next, the user enters a new value (e.g., *Robert*) for the identifier of the model element (e.g., the *name*) in a pop-up dialog box, and then the model element's identifier is refactored in an integrated manner across all diagrams and textual expressions. Note that selecting a part of the textual expression that does not represent an existing graphical model element (e.g., *Charles*) will not allow the user to carry out any refactoring.

F. Consistency Enforcement

Consistency is automatically enforced within the hybrid DSL editor when referenced model elements in diagrams are

renamed or deleted. This avoids unnecessary inconsistency in the model by maintaining consistency between the graphical and textual parts of the model. 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 are updated accordingly. For instance, when in the diagram, the *person* named *Bob* is renamed to *Robert*, then in the smart textual editor, the third line from the *efforts* textual expression will be updated accordingly by replacing *Bob* with *Robert*. In addition, if *Bob* is deleted in the diagram, then the reference from the third line of the textual expression will be removed as well.

G. Uniform Error Reporting

Graphite stores in memory the diagnostics information, i.e., the errors that are produced when parsing a syntactically incorrect textual expression and when reference resolution fails. When a validation operation is triggered in the hybrid DSL editor, these diagnostics are used to populate error markers in the Problems view. The Problems view from the running example reports an error message informing the user that the *efforts* textual expression could not be parsed due to the unresolved reference to *Charles*.

H. Conditional Storage of Derived Model Elements

When the model is loaded, all textual expressions are parsed according to their associated grammars, and for each of them, the derived model elements are assigned to their respective properties specified by the annotations. For example, when the model is loaded, *effortsExpression* is parsed and the derived model elements are assigned to the *efforts* list. Graphite stores on disk the derived model elements (i.e., the *efforts* list) when the model resource is saved, but only in the case that the textual expression (i.e., the *effortsExpression* string) is in an invalid state, i.e., it cannot be parsed successfully or reference resolution fails. Therefore, the derived model elements are stored on disk if the textual expression is in an invalid state. However, if it is in a valid state, the derived model elements are not stored on disk, as they can be recovered when the model resource is loaded. For example, the *efforts* list containing four *Effort* model elements in Figure 1 is stored on disk, as *effortsExpression* is in an invalid state.

I. Metamodel and Grammar Validation

The grammar from Figure 1 defines the 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. Grammars must adhere to the structure of the annotated metamodel, taking into account the annotation(s). Specifically, the entry rule must derive a root model element that is an instance of the meta-class (e.g., *Task*) that contains the property being expressed through a textual syntax (e.g., *efforts*). Furthermore, only the property being expressed through a textual syntax needs to be populated by the grammar's entry rule (e.g., only the *efforts* property of *Task*).

To assist language engineers, Graphite includes a validation mechanism that checks whether a given grammar is compliant with the structure of the annotated metamodel. For validating a grammar, Graphite carries out a model validation operation that takes as input the grammar and the annotated metamodel. Additionally, Graphite provides the capability to validate the annotated metamodel, ensuring that the added annotations are properly defined and that the annotation attributes match existing properties within the metamodel (i.e., *feature* and *derive*), the grammar's name (*grammar*), and the grammar's entry rule (*entryRule*).

J. Grammar Generation

Graphite requires a grammar that conforms to the specific structural constraints of the annotated metamodel, as outlined in the previous subsection. To streamline and simplify the language engineering process, Graphite offers functionality for automatically generating the initial structure (the skeleton) of grammars that are compliant with its constraints. These generated grammars serve as a starting point for the language engineer, who can further customise and refine them as needed. This capability is realised through a model-to-text transformation which takes the annotated metamodel as input and produces one or more compliant grammars, based on the grammars referenced by the annotations.

K. Generation of Hybrid DSL Editors

The principal capability provided by Graphite is the automatic generation of hybrid DSL editors. Graphite carries out a model-to-text transformation that takes as input the annotated metamodel, the generator model (the *.genmodel* file), the Sirius VSM, and the Xtext grammar(s). The transformation generates implementation code for configuring hybrid DSL editors. The generated hybrid DSL editors provide all capabilities discussed in this section.

V. Language Engineering Process: Demonstration

Figure 2 presents the language engineering process one would have to follow using Graphite to develop a hybrid DSL editor as the one in Figure 1. The process begins with the language engineer defining the metamodel of the DSL, as in Listing 2. Next, the language engineer decides which parts of the metamodel should be expressed through a textual syntax. In the context of the running example, it is preferred to capture the *efforts* using an embedded textual notation. Accordingly, in the second step, the metamodel is extended by adding a string attribute (i.e., *effortsExpression*) that stores the textual representation of the *efforts*, and an annotation that links the added string attribute with the grammar (the grammar's name) to which it must conform. The output of the second step is the annotated metamodel presented in Listings 1 and 2. Note that for following the exact steps described in this paper, one has to convert the initial metamodel into Emfatic format (*right-click* in Eclipse and select "*Generate Emfatic Source*"), and then convert the Emfatic representation back into an Ecore metamodel (*right-click* and select "*Generate Ecore Model*").

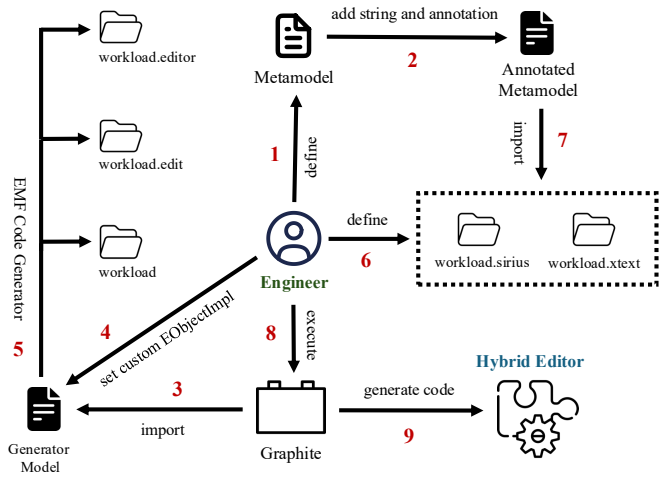

Fig. 2. Language Engineering Process with Graphite

In the third and fourth steps, Graphite is imported into the project (i.e., the *workload* project) containing the generator model; the language engineer modifies the generator model by setting the "*Root Extends Class*" property to a custom *EObjectImpl* class [16] from Graphite. This will change the superclass of the *EObject* classes (i.e., *EffortImpl*, *TaskImpl*, *PersonImpl*) that will be generated by the EMF code generator. Next, the EMF code generator is launched through the generator model to generate the model code (i.e., the *workload* project), edit code (i.e., the *workload.edit* project) and editor code (i.e., the *workload.editor* project).

In the sixth step, the language engineer defines the graphical part of the syntax in a Sirius VSM (i.e., the *workload.sirius* project), as in the left side of Figure 1, to specify that *Person* model elements should be represented using "*person.png*", *Task* model elements using "*task.png*" and the *Dependencies* and *Leader* edges as a solid edge. Additionally, the language engineer defines the textual part of the syntax through the Xtext grammar (i.e., the *workload.xtext* project) presented on the right side of Figure 1. The grammar specifies that each *Effort* is separated by a newline, containing a *person* and a number of *months* separated by a colon. In the seventh step, both the Sirius VSM and the Xtext grammar import the annotated metamodel, to be able to reference the parts of the metamodel they operate over.

Finally, the language engineer triggers the code generation facility provided by Graphite, to automatically generate the code required for configuring a hybrid DSL editor. One must select in the Eclipse Workspace the annotated metamodel (*.ecore*), the generator model (*.genmodel*), the Sirius VSM (*.odesign*) and the Xtext grammar (*.xtext*), then *right-click* and select in the context menu "*Graphite > Generate Hybrid Editor*". Before performing code generation, Graphite first validates the selected metamodel and grammar (see Section IV-I), and reports any errors or warnings in the Problems view.

VI. Conclusions and Future Work

This paper presented Graphite, a tool that aids the automated development of hybrid DSL editors. For generating a hybrid DSL editor, Graphite requires as input an Ecore metamodel, an EMF generator model, a Sirius VSM, and one or more Xtext grammars. The paper outlined the capabilities provided by Graphite and demonstrated its language engineering process.

In future work, we plan to investigate unified capabilities for searching and for finding all references across the textual and graphical parts of the model.

Acknowledgments

The work in this paper has been partially funded through the SCHEME InnovateUK project (contract no. 10065634).

References

[1] J. Cooper and D. Kolovos, "Engineering Hybrid Graphical-Textual Languages with Sirius and Xtext: Requirements and Challenges," in *ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 2019, pp. 322–325.

[2] I. Predoaia, D. Kolovos, M. Lenk, and A. García-Domínguez, "Streamlining the Development of Hybrid Graphical-Textual Model Editors for Domain-Specific Languages," *Journal of Object Technology*, vol. 22, no. 2, 2023.

[3] I. Predoaia, "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, 2023, pp. 153–158.

[4] I. Predoaia, D. Kolovos, and A. Garcia-Dominguez, "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*, 2024, pp. 1156–1160.

[5] I. Predoaia, D. Kolovos, and A. García-Dominguez, "Reimplementing the Structurizr Software Architecture Modelling Language as a Hybrid DSL," in *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2025, pp. 380–386.

[6] I. Predoaia, J. Harbin, S. Gerasimou, C. Vasiliou, D. Kolovos, and A. García-Domínguez, "Tree-Based versus Hybrid Graphical-Textual Model Editors: An Empirical Study of Testing Specifications," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, 2024, pp. 80–91.

[7] Eclipse, *Specifying Sirius Viewpoints*, [Online]. Available: https://eclipse.dev/sirius/doc/4.0.x/specifier/general/Specifying_Viewpoints.html, (Last Accessed: 2025-07-10).

[8] JetBrains, *JetBrains MPS Website*, [Online]. Available: https://www.jetbrains.com/mps, (Last Accessed: 2025-07-10).

[9] M. Scheidgen, "Textual Modelling Embedded into Graphical Modelling," in *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 2008, pp. 153–168.

[10] Obeo and TypeFox, *Xtext Sirius integration - white paper*, [Online]. Available: https://www.obeodesigner.com/resource/white-paper/WhitePaper_XtextSirius_EN.pdf, (Last Accessed: 2025-07-10).

[11] J. Cooper, "A Framework to Embed Textual Domain Specific Languages in Graphical Model Editors," Master's thesis, University of York, 2018.

[12] Altran, *Xtext Sirius integration*, [Online]. Available: https://altran-mde.github.io/xtext-sirius-integration.io, (Last Accessed: 2025-07-10).

[13] F. Ciccozzi, M. Tichy, H. Vangheluwe, and D. Weyns, "Blended Modelling - What, Why and How," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 2019, pp. 425–430.

[14] Eclipse, *Emfatic*, [Online]. Available: https://eclipse.org/emfatic, (Last Accessed: 2025-07-10).

[15] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "The Epsilon Object Language (EOL)," in *Model Driven Architecture – Foundations and Applications*, A. Rensink and J. Warmer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 128–142.

[16] Eclipse, *EObjectImpl*, [Online]. Available: https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/impl/EObjectImpl.html, (Last Accessed: 2025-07-10).