White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
**Universities of Leeds, Sheffield and York**

Deposited via The University of York.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/208677/

Version: Accepted Version

**Proceedings Paper:**
Predoaia, Ionut (2023) Towards Systematic Engineering of Hybrid Graphical-Textual Domain-Specific Languages. In: Proceedings - 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS-C 2023. 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS-C 2023, 01-06 Oct 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS). IEEE, SWE, pp. 153-158.

https://doi.org/10.1109/MODELS-C59198.2023.00041

**Reuse**
This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here: https://creativecommons.org/licenses/

**Takedown**
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

UNIVERSITY OF LEEDS University of Sheffield UNIVERSITY of York

# Towards Systematic Engineering of Hybrid Graphical-Textual Domain-Specific Languages

Ionut Predoaia
*Department of Computer Science*
*University of York*
York, United Kingdom
ionut.predoaia@york.ac.uk

***Abstract*—A domain-specific language (DSL) can have graphical, textual or hybrid syntaxes. Certain domain concepts are better suited to be represented graphically, whereas a textual representation is often more appropriate for modeling the behavior, complex expressions, and fine details of a domain. As such, the best of both worlds of graphical and textual modeling can be delivered by a DSL that has a hybrid (part-graphical and part-textual) syntax. The engineering of hybrid graphical-textual DSLs and their supporting workbenches is a non-trivial endeavor, as a substantial amount of hand-written code is required. Existing workbenches for hybrid graphical-textual DSLs pose several limitations, e.g., they do not enforce the consistency between the graphical and textual parts of the model and do not provide uniform error reporting. This work will propose a methodology and provide tooling for systematic engineering that aims to minimize the accidental complexity involved in designing and developing hybrid graphical-textual DSLs and their supporting workbenches.**

***Index Terms*—Domain-Specific Languages, Graphical-Textual Modeling, Language Engineering, Model Editors**

## I. Introduction

Graphical syntaxes are convenient for the simplification of the high-level concepts of a system, and are efficient at representing graph-like structures, however, it can often be tedious and error-prone to capture and maintain graphically fine details such as precise behavior and complex expressions. Furthermore, graphical syntaxes can reduce the time spent linking model elements together, whereas textual syntaxes can reduce the number of clicks when creating and editing models [1]. As such, textual syntaxes can complement graphical syntaxes to deliver the best of both worlds of graphical and textual modeling.

Hybrid graphical-textual DSLs and their supporting workbenches can surely be developed with projectional editors such as JetBrains MPS, however, projectional editors do not provide a pure textual editing experience and they are often perceived as problematic, due to the unfamiliar editing experience and challenges in the integration with existing infrastructure [2]. Accordingly, the intent of this work is to use non-projectional editors, i.e., parser-based editors for textual sub-syntaxes. This work is focused on languages that are predominantly graphical, but which would benefit from embedded textual sub-syntaxes to define complex expressions or behavior. This is an often encountered pattern in interactions with industrial collaborators, e.g., the Rolls-Royce CaMCOA DSL [3] and state machines that provide a graphical syntax for states and transitions and a textual syntax for guards and actions.

## II. Terminology

The term *hybrid graphical-textual syntax* is used to refer to a part-graphical and part-textual syntax. In addition, the term *hybrid graphical-textual language* is used to refer to a language that has a hybrid graphical-textual syntax. When using a hybrid graphical-textual 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* is used to refer to the textual parts of the model. The term *textual expression* is used to refer to any text that conforms to the syntax specified by a formal grammar. The term *model editor* is used rather than the term *workbench*, as a model editor is included in a workbench. A workbench covers a broader scope, as it contains a wide range of components, e.g., menus, toolbars, perspectives [4].

## III. Problem

A small number of prior research efforts have been carried out to facilitate the engineering of hybrid graphical-textual DSLs and their supporting model editors. The past research works have rather focused on DSLs that have both a graphical syntax and a textual syntax, that are designed and used independently of each other. Several different notations are provided for the same concepts of the abstract syntax, and users can choose to use the syntax that is preferred or that is most appropriate for editing specific model elements. The DSLs used in the prior works are not hybrid graphical-textual DSLs, as they do not have a part-graphical and part-textual syntax, but rather they have a graphical syntax and a textual syntax, both being independent of each other.

A limited amount of works target the same line of research as the one presented in this paper, and they are summarized in Section VI. They address hybrid graphical-textual DSLs, as defined in this paper. However, the DSLs are not systematically engineered and their supporting model editors pose several open challenges.

When developing a hybrid graphical-textual DSL, a language designer must specify which parts of the language's syn-

tax are graphical and which parts are textual. For this purpose, one would ideally use a declarative specification for aiding the language engineer in the definition of the language's syntax. For defining the graphical parts of the language's syntax, one should declaratively associate graphical representations with some parts of the abstract syntax. Similarly, for defining the textual parts of the language's syntax, one should declaratively associate parts of the abstract syntax with one or more formal grammars to which they must conform. In the past solutions, the graphical parts of the language's syntax can be defined via a declarative specification, e.g., via a Sirius Viewpoint Specification Model (VSM) [5]. However, no techniques have been proposed for defining the textual parts of the language's syntax via a declarative specification.

**Q1** How to specify which parts of the language's syntax are graphical and textual by using a declarative specification?

The development of state-of-the-art hybrid graphical-textual model editors for DSLs is based on hand-written code that integrates custom textual model editors into graphical model editors. For each property from the metamodel (i.e., abstract syntax) that is expressed through a textual syntax, hand-written glue code is required for embedding a custom textual model editor into the graphical model editor. This solution design is prone to human errors, and can largely be automated by leveraging model-driven engineering techniques that generate the required glue code automatically.

**Q2** How can the development of hybrid graphical-textual model editors be streamlined?

Hybrid graphical-textual model editors from prior works do not provide consistency checking or enforcement as the model evolves, across the graphical and textual parts of the model. A model editor should enforce consistency between the graphical and textual parts of the model, e.g., when a graphical model element is deleted from the diagram, then the textual expressions that were referencing the graphical model element must be updated accordingly, to reflect the lost reference.

**Q3** How to maintain consistency across the graphical and textual parts of the model?

Error reporting has not been thoroughly addressed by state-of-the-art hybrid graphical-textual model editors. They provide error reporting capabilities that are either not uniform or are not efficient. Model inconsistencies should be uniformly and transparently reported as errors, such that users can navigate to problematic model elements from a reported error. An example of model inconsistency is when a textual expression refers to a graphical model element that no longer exists in the diagram.

**Q4** How can the model editors report model inconsistencies uniformly and efficiently?

## IV. Requirements for Hybrid Graphical-Textual Model Editors

This section presents the required capabilities for hybrid graphical-textual model editors that have been motivated by prior works with industrial partners [1], [6].

**R1** The textual editors must support syntax-aware editing features, such as syntax highlighting, auto-completion, refactoring, and error detection markers.

**R2** The textual expressions must be able to reference model elements that have been defined graphically in the diagram. Furthermore, one should be able to navigate from a textual expression to a referenced graphical model element in the diagram.

**R3** As the model evolves, the model editor must automatically enforce the consistency of the references between the textual and graphical parts of the model. For instance, when a graphical model element is deleted from the diagram, any textual expressions that were referencing the deleted graphical model element must be updated accordingly, to reflect the lost reference.

**R4** The model editor must uniformly report inconsistencies from the textual and graphical parts of the model as errors. One should be able to navigate to the problematic model element from a reported error.

**R5** The model that is expressed through a hybrid graphical-textual syntax must be exposed to model management programs (e.g., model-to-model and model-to-text transformations, model validation operations) as a single unified abstract syntax graph (ASG) that integrates elements from both the textual and graphical parts of the modeled system. The textual expressions must not be exposed to model management programs as plain text, but rather as model element(s).

## V. Motivating Example

This section presents a minimal contrived example [6] to demonstrate the motivation of this work while keeping accidental complexity to a minimum. Note that the complete metamodel of the example is presented in [6].

Listing 1 presents the metamodel of a DSL for modeling project plans, that has been defined in Emfatic [7]. Note that Emfatic is a convenient textual syntax for Ecore metamodels. The metamodel specifies that a *Project* contains a list of *tasks* and a list of *people*. A *Task* has a *name* and a list of *efforts*. Each *Effort* is assigned to a *person* and has a number of *months*. For the purpose of this example, it is preferred to use a textual syntax for specifying the *efforts*.

Figure 1 displays a modeled *project* in a hybrid graphical-textual model editor, in which the *tasks* and the *people* are modeled graphically, whereas the *efforts* are expressed with a YAML-like textual syntax, for which each *effort* is specified on a separate line, as a key-value pair in the form {*person*}:{*months*}. Therefore, the *tasks* and the *people* model elements represent the graphical parts of the model, whereas the *efforts* represent the textual parts of the model. The *task* named *Implementation* is selected in the diagram, therefore its properties (i.e., the *name* and the *efforts*) are displayed in the properties view. Each line from the *efforts* textual expression represents an *effort* model element, e.g., the first line is an *effort* that refers to a *person* named *Alice* and has

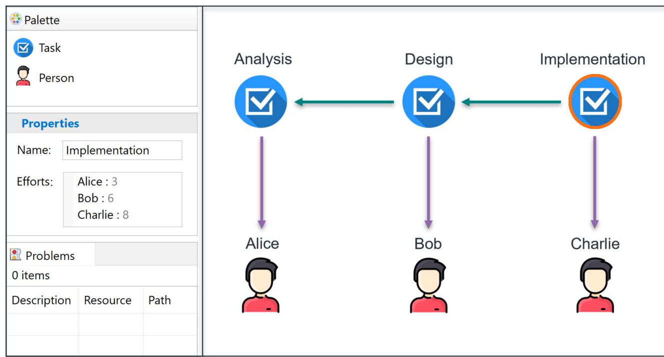


Fig. 1. Hybrid Graphical-Textual Model Editor [6]

```
package workload;

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

class Task {
  attr String name;
  val Effort[*] efforts;
}

class Person {
  attr String name;
}

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

Listing 1. Metamodel of the Project Workloads DSL

a value of 3 *months*. The *efforts* textual expression references graphical model elements, i.e., *person* model elements that are modeled graphically in the diagram. It is desirable for the textual editor that contains the textual expression to support developer assistance features, such as syntax highlighting, auto-completion, and error detection. The envisioned model editor should automatically enforce consistency between the textual and graphical parts of the model. For instance, when the *person* named *Alice* is renamed into *David* in the diagram, then the first line from the textual expression would be updated, by replacing *Alice* with *David*. The first line from the textual expression references the *person* named *Alice*, however, if the person named *Alice* was not graphically defined in the diagram, an error would have been reported in the problems view. Users only see and modify the textual representation of the *efforts*, however, it is desirable that they interact behind the scenes with a concrete instance of the *efforts* list from the metamodel (i.e., `val Effort[*] efforts`). The *efforts* textual expression is parsed whenever it is modified, and resulting *effort* model elements are derived. Accordingly, it is desirable to have the textual representation of the *efforts* exposed to model management programs as a list of *effort* model elements, rather than as plain text.

## VI. Related Work

There are four main works [8]–[11] that follow the same line of work presented in this paper. These works address hybrid graphical-textual DSLs and their supporting model editors that meet several of the requirements presented in Section IV. Other related works are based on blended modeling [12] and on Capella [13], however, these two approaches do not target hybrid graphical-textual DSLs, as justified in [6].

A technique for embedding textual modeling into graphical modeling has been presented in [8]. The solution is based on the integration of the Textual Editing Framework (TEF) and Graphical Modeling Framework (GMF). In [8], parts of the abstract syntax are associated with a graphical syntax, and one can view the textual representation of a graphical model element in a TEF textual editor. Note that TEF editors provide developer assistance features, such as syntax highlighting, code completion, and error markers. This approach has the advantage that one can have a high-level graphical representation of a domain concept and can view the low-level details of the same domain concept using a textual representation. References between the textual expressions and the graphical model elements are supported, however, techniques for reporting model inconsistencies as errors are not addressed. When the textual representation is modified and saved, the changes are committed and merged with the underlying graphical model element. Therefore, the modeled system can transparently be exposed to model management programs as a unified ASG.

Obeo [14] and Typefox [15] have presented two case studies on the integration of Xtext and Sirius in [9]. In the second case study, users can edit the model graphically and textually from within the same model editor. An Xtext textual editor has been embedded into the Sirius graphical model editor. The graphical model elements have textual properties that contain textual expressions. When a graphical model element is selected in the diagram, the textual editor is displayed in the properties view. The textual expressions can be written using syntax-aware editing features and can reference graphical model elements defined in the diagram. Additionally, one can navigate from the textual editor to a referenced graphical model element in the diagram. Errors are only reported when the textual editor associated with a textual expression is being displayed. Otherwise, if the textual editor is no longer displayed, then the user would not be aware of whether an error exists in the textual expression. It can be concluded that errors are not reported uniformly. Moreover, model management programs only see the textual expressions as plain text, rather than as model elements.

Further research has been carried out in [10] with the aim of embedding textual DSLs into graphical model editors. The textual expressions can be written using developer assistance features and can reference graphical model elements defined in the diagram. Error reporting has been realized uniformly, but inefficiently, by implementing a custom builder that is executed each time the project is built, parsing all textual expres-

sions to identify potential errors and report them. Furthermore, the textual expressions are exposed to model management programs as plain text, instead of model elements.

Capgemini Engineering [16] (previously known as Altran) has extended the work from [9] by embedding Xtext textual editors into Sirius graphical diagrams, in addition to the properties view [11]. The textual expressions can be written using developer assistance features and can reference graphical model elements. Errors are not reported in the problems view, however, they are reported in a pop-up window whenever users type syntactically incorrect textual expressions. The textual editors that are embedded in the diagrams display the textual representation of the underlying model element. Whenever one modifies a textual expression in a textual editor that is embedded in the diagram, the textual expression is parsed and the derived model element(s) is merged with the underlying model element. Therefore, the textual expressions can transparently be exposed as model elements to model management programs.

None of the prior works propose consistency enforcement techniques to ensure that as the model evolves, consistency is maintained across the graphical and textual parts of the model. Furthermore, the prior solutions are realized via a considerable amount of lines of hand-written code. Moreover, a declarative specification is used in the 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.

## VII. Proposed Solution

Considering that all prior works that follow the same line of work are based on EMF, it has been decided to build on the existing works [9]–[11], by proposing a solution based on EMF as well. Therefore, EMF-based language workbenches have been chosen as the basis for this research project, i.e., Sirius and Xtext.

The requirements **R1** and **R2** can conveniently be fulfilled with the out-of-the-box facilities provided by Xtext. To address **R1**, Xtext will be used to define textual syntaxes and to generate their supporting textual editors that provide syntax-aware editing features. Note that with an Xtext grammar, one can derive an Ecore-based metamodel for representing the abstract syntax of the language, or can import an existing Ecore-based metamodel of a language. In this case, the latter is used, as the purpose of the grammar is to define the textual syntax of various parts from an existing metamodel. An Xtext grammar will be defined, that will import the metamodel of the Project Workloads DSL, and will define a YAML-like textual syntax for the *efforts* property. For addressing **R2**, the grammar must declare rules that define cross-references to model element types from the metamodel that will be expressed with a graphical syntax. As such, the language engineer must be aware of the model element types that will be expressed with a graphical syntax and then declare grammar rules that define references to those specific types. For instance, the grammar that specifies the textual syntax of the *efforts* must declare a rule that defines a reference to model elements of type *Person*.

To address **R5**, one option is to generate a textual projection whenever the underlying model elements are accessed. The textual expressions are not stored but rather generated at runtime based on the underlying model elements. The advantage of this approach is that the underlying model elements of the textual expression can be directly accessed by model management programs. However, by using this technique, temporary inconsistencies are not tolerated by the model editor, as justified in [6]. For the purpose of tolerating temporary inconsistencies [6], it has been decided to modify the metamodel, such that for each property that will be expressed with a textual syntax, an additional string attribute will be added to the metamodel that will record the textual expressions. Consequently, each property that is expressed with a textual syntax is associated with a string attribute that contains an equivalent projection of its textual representation, whilst the property and the string attribute are bidirectionally synchronized. For instance, a string attribute called *effortsExpression* will be added to the metamodel, which will record the textual representation of the list of *efforts*. The bidirectional synchronization will be realized via serialization and deserialization, i.e., each time the list of *efforts* is modified, it is serialized as text that will replace the current value of *effortsExpression*, whereas each time the textual expression is modified, it is parsed and the derived *effort* model elements will overwrite the content of the *efforts* property. A property that is expressed through a textual syntax (e.g., the *efforts* property) will be called a derived property, for the reason that it contains the model elements that are derived from parsing the string attribute containing the textual expression.

In regard to addressing **Q1**, a Sirius VSM will be used for declaratively specifying which parts of the abstract syntax are associated with a graphical representation. Additionally, the metamodel will be annotated by defining annotation mappings between a derived property, its associated string attribute that contains an equivalent textual representation, and the grammar that specifies the textual syntax to which the textual representation must conform. For instance, an annotation will be added to the metamodel that will specify that the *effortsExpression* property must be parsed using the defined grammar for the YAML-like textual syntax, and that it represents an equivalent textual projection of the *efforts* list. The annotated metamodel will be passed as input to a model-to-text transformation that will generate the glue code for the bidirectional synchronization of the derived properties and their associated textual expressions, and for embedding a textual editor with assistance features in the Sirius properties view for each derived property.

For addressing **Q3** and **R3**, each time a graphical model element that is referenced by a textual expression changes its identifier, then the content of the derived property must be serialized as a textual expression that will overwrite the content of the corresponding string attribute. To realize this behavior, an efficient technique would be to identify the

references from a textual expression during parsing, and attach event listeners to each reference. For instance, when parsing the textual expression from the motivating example, three references would be identified, i.e., the *person* objects named *Alice*, *Bob*, and *Charlie*, and an event listener will be attached to each *person*. When a referenced model element changes its identifier, the event listener will trigger the serialization of the content of the derived properties that are referencing the model element, and the resulting serialized text will overwrite the content of the associated string attributes. For example, when the *person* named *Alice* is renamed into *David*, an event listener will trigger the serialization of the *efforts* into a textual expression that will overwrite the content of *effortsExpression*.

In regard to addressing **Q4** and **R4**, the diagnostics information will be stored in memory, i.e., the errors that are produced when parsing syntactically incorrect textual expressions and when reference resolution fails. When a validation operation is triggered in the model, then the diagnostics information that is stored in memory will be used to populate error markers in the problems view. This approach is efficient only when applied in conjunction with the consistency enforcement techniques that have been proposed for addressing **Q3**. By applying the consistency enforcement techniques, the textual expressions are parsed only when necessary. Therefore, this solution is efficient, as opposed to the technique from [10], where a custom builder parses all textual expressions each time the project is built to report errors, although no errors may exist.

Finally, for addressing **Q2**, a framework will be developed that provides facilities for the fulfillment of all requirements presented in Section IV. Via model transformations that take the declarative specification(s) as input, code that delegates API calls to the framework will be generated automatically. By using this technique, one would require little or no hand-written code for the development of hybrid graphical-textual model editors.

## VIII. Expected Contributions

The expected contributions of this work are:

- a methodology for systematic engineering of hybrid graphical-textual DSLs and their supporting model editors;
- automated facilities and tools that streamline the development of hybrid graphical-textual model editors for DSLs;
- hybrid graphical-textual DSLs and their supporting model editors implemented using the developed facilities;
- guidelines for applying the methodology with language workbenches, other than Sirius and Xtext.

## IX. Plan for Evaluation and Validation

The methodology for the systematic engineering of hybrid graphical-textual DSLs and their supporting model editors will be evaluated through case studies from industry and academia.

A full-scale case study is provided by NetApp [17], a global software company that delivers hybrid cloud data services and data management services. A hybrid graphical-textual DSL and its supporting model editor will be developed for modeling NetApp public cloud services, to lower the entry barrier to cloud services adoption. The DSL uses a graphical syntax for simplification of high-level infrastructure components, and textual syntaxes for defining low-level details. Another case study will involve developing a hybrid graphical-textual DSL and its supporting model editor in the context of the SESAME project [18]. The DSL specifies the testing space for testing multi-robot systems (MRS) in simulation, via a graphical syntax for defining the high-level elements of the testing space, and via a textual syntax for defining simulation-specific conditions. Additional case studies could involve OCLInEcore [19] and the RecordFlux [20] protocol specification language. Moreover, user studies will be carried out, in which the average language engineer will assess the usability of the supporting tooling that will be provided for the systematic engineering of hybrid graphical-textual DSLs.

The following criteria will be evaluated:

- **Completeness and Correctness**. Test cases will be used for assessing completeness and correctness (e.g., for bidirectional synchronization between the derived properties and their associated textual expression, for checking that consistency is maintained as the model evolves, for checking that errors are reported in the case of model inconsistencies) by measuring the percentage of passed test cases.
- **Amount of effort**. The amount of effort (e.g., the number of lines of hand-written code) for meeting the requirements will be compared against state-of-the-art solutions.
- **Parse Operations**. The number and total duration times of executed parse operations for applying the proposed consistency enforcement and uniform error reporting techniques will be compared against naive and inefficient approaches.
- **Scalability**. The execution time of various events (e.g., loading and storing the model) will be measured in the case of an increasing number of textual expressions.

## X. Current Status

A paper [6] has been published based on the progress to date. The work proposes Graphite [21], a tool that streamlines the development of hybrid graphical-textual model editors (**Q2**) based on Sirius and Xtext, whilst fulfilling the requirements from Section IV. The solution has been evaluated in the industrial case study provided by NetApp. Furthermore, no experimental evaluations related to performance aspects have been carried out. Moreover, the above work has partially addressed the definition of a declarative specification (**Q1**), consistency enforcement techniques (**Q3**), and uniform and efficient error reporting (**Q4**), however, no evaluation has yet been conducted.

## Acknowledgment

The work in this paper has been funded through NetApp, the HICLASS InnovateUK project (contract no. 113213), and the European Union's Horizon 2020 project SESAME (grant agreement no. 101017258). I would like to thank Prof. Dimitris Kolovos, Dr. Antonio García-Domínguez and Dr. Matthias Lenk for their supervision in this work.

# 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] M. Voelter, J. Siegmund, T. Berger, and B. Kolb, "Towards User-Friendly Projectional Editors," in *International Conference on Software Language Engineering*. Springer, 2014, pp. 41–61.

[3] J. Cooper, A. De la Vega, R. Paige, D. Kolovos, M. Bennett, C. Brown, B. S. Pina, and H. H. Rodriguez, "Model-Based Development of Engine Control Systems: Experiences and Lessons Learnt," in *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2021, pp. 308–319.

[4] E. Burnette, *Eclipse IDE Pocket Guide: Using the Full-Featured IDE*. "O'Reilly Media", 2005.

[5] Eclipse, *Specifying Viewpoints*, [Online]. Available: https://eclipse.dev/sirius/doc/4.0.x/specifier/general/Specifying_Viewpoints.html, (Last Accessed: 2023-07-20).

[6] 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.

[7] Eclipse, *Emfatic*, [Online]. Available: https://eclipse.org/emfatic, (Last Accessed: 2023-07-20).

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

[9] Obeo and TypeFox, *Xtext Sirius integration - white paper*, 2017, [Online]. Available: https://www.obeodesigner.com/resource/white-paper/WhitePaper_XtextSirius_EN.pdf, (Last Accessed: 2023-07-20).

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

[11] Altran, *Xtext Sirius integration*, [Online]. Available: https://altran-mde.github.io/xtext-sirius-integration.io, (Last Accessed: 2023-07-20).

[12] L. Addazi and F. Ciccozzi, "Blended graphical and textual modelling for UML profiles: A proof-of-concept implementation and experiment," *Journal of Systems and Software*, vol. 175, p. 110912, 2021.

[13] Eclipse Foundation, *Capella Textual Editor Extension*, [Online]. Available: https://github.com/eclipse/capella-textual-editor, (Last Accessed: 2023-07-20).

[14] Obeo, *Homepage*, [Online]. Available: https://www.obeosoft.com, (Last Accessed: 2023-07-20).

[15] Typefox, *Homepage*, [Online]. Available: https://www.typefox.io, (Last Accessed: 2023-07-20).

[16] Capgemini, *Homepage*, [Online]. Available: https://www.capgemini.com, (Last Accessed: 2023-07-20).

[17] NetApp, *Homepage*, [Online]. Available: https://www.netapp.com, (Last Accessed: 2023-07-20).

[18] SESAME, *Homepage*, [Online]. Available: https://www.sesame-project.org, (Last Accessed: 2023-07-20).

[19] Eclipse, *OCLinEcore*, [Online]. Available: https://wiki.eclipse.org/OCL/OCLinEcore, (Last Accessed: 2023-07-20).

[20] AdaCore, *RecordFlux Protocol Specification DSL*, [Online]. Available: https://github.com/AdaCore/RecordFlux, (Last Accessed: 2023-07-20).

[21] I. Predoaia, *Graphite*, [Online]. Available: https://github.com/epsilonlabs/graphite, (Last Accessed: 2023-07-20).