UNIVERSITY of York

This is a repository copy of *Streamlining the Development of Hybrid Graphical-Textual Model Editors for Domain-Specific Languages*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/202852/

Version: Published Version

Article:

Predoaia, Ionut orcid.org/0000-0002-2009-4054, Kolovos, Dimitris orcid.org/0000-0002-1724-6563, Lenk, Matthias et al. (1 more author) (2023) Streamlining the Development of Hybrid Graphical-Textual Model Editors for Domain-Specific Languages. Journal of Object Technology. ISSN 1660-1769

https://doi.org/10.5381/jot.2023.22.2.a8

Reuse

This article is distributed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivs (CC BY-NC-ND) licence. This licence only allows you to download this work and share it with others as long as you credit the authors, but you can't change the article in any way or use it commercially. 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.

White Rose
university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Streamlining the Development of Hybrid Graphical-Textual Model Editors for Domain-Specific Languages

Ionut Predoaia[*], **Dimitris Kolovos**[*], **Matthias Lenk**[‡], **and Antonio García-Domínguez**[*]
[*]University of York, United Kingdom
[‡]NetApp, Germany

ABSTRACT A domain-specific language (DSL) can have multiple syntaxes, that can be graphical or textual. When modelling the behaviour, complex expressions, and fine details of a domain, often it does not make sense to use a graphical syntax, as this can result in large, crowded diagrams, therefore in such cases, a textual syntax is often more appropriate. As such, the best of both worlds can be delivered by a DSL that has hybrid (part-graphical and part-textual) syntaxes. In this work, we address open challenges and apply model-driven engineering techniques to streamline the development of hybrid graphical-textual model editors for DSLs, by using as little hand-written code as possible.

KEYWORDS Domain-Specific Language, Model Editor, Graphical-Textual Modelling, Language Engineering, Code Generation, EMF, Sirius, Xtext.

1. Introduction

A language, domain-specific or not, has an abstract syntax and one or more concrete syntaxes. The abstract syntax of a language defines the language concepts, the relationships between the concepts and the constraints, independent of any representation. The concrete syntax of a language defines how the concepts of the abstract syntax are presented to the language user. Concrete syntaxes can be graphical or textual. Textual syntaxes use textual tokens following a grammar of syntactic rules to express domain abstractions, whereas graphical syntaxes use diagrams, visual notations, and symbols to represent domain concepts. Diagrams are better suited to represent the high-level concepts of a system, as it can often be tedious and error-prone to define and maintain graphically the fine details of a system, such as precise behaviour, complex expressions, actions, or documentation, therefore for such cases textual syntaxes are more appropriate (Cooper & Kolovos 2019). Hybrid graphical-textual model editors 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 (Cooper & Kolovos 2019).

State-of-the-art hybrid graphical-textual model editors pose several open challenges and their development is a non-trivial endeavour as a large amount of hand-written code is required. Our main contribution is Graphite, a tool that can be leveraged to streamline the development of hybrid graphical-textual model editors for DSLs, with as little hand-written code as possible. It is important to note that hybrid graphical-textual model editors can surely be developed with projectional editors such as JetBrains MPS[1]. However, projectional editors do not provide a pure textual editing experience and they are often perceived as problematic. Therefore, we are interested in making use of non-projectional editors, i.e., parser-based editors for textual sub-syntaxes. Accordingly, the envisioned graphical-textual model editors are EMF-based, and rely on the integration of the Sirius graphical modelling framework and the Xtext textual modelling framework. Our focus is on languages that are predominantly graphical, but which would benefit from embedded textual sub-syntaxes to define complex expressions or

JOT reference format:
Ionut Predoaia, Dimitris Kolovos, Matthias Lenk, and Antonio García-Domínguez. *Streamlining the Development of Hybrid Graphical-Textual Model Editors for Domain-Specific Languages*. Journal of Object Technology. Vol. 22, No. 2, 2023. Licensed under Attribution - NonCommercial - No Derivatives 4.0 International (CC BY-NC-ND 4.0) http://dx.doi.org/10.5381/jot.2023.22.2.a8

[1] https://www.jetbrains.com/mps

behaviour. This is a pattern that we have often encountered in interactions with industrial collaborators, particularly where full code generation is required.

The remainder of this paper is organised as follows. Section 2 presents the envisioned hybrid graphical-textual model editors. Section 3 introduces the background of our work. Section 4 discusses related work. Section 5 presents the contributions of this paper. Section 6 evaluates our contributions. Section 7 outlines the limitations of our work. Finally, Section 8 concludes the paper and provides directions for future research.

2. Hybrid Graphical-Textual Model Editors

This section presents the capabilities that a modeller would typically require and expect from a hybrid graphical-textual model editor. The envisioned hybrid graphical-textual model editors are composed of a diagram, embedded textual editor(s) with assistance features, a view in which one can edit the properties of a selected model element, a view with various symbols that can be dragged and dropped on top of the diagram to create model elements of different types, and a view that displays reported errors from the model.

2.1. Terminology

In this section, we define various terms that are used throughout this paper. 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, and the term *textual model elements* is used to refer to the textual parts of the model.

2.2. Motivating example

Listing 1 presents the abstract syntax, i.e., the metamodel of a minimal contrived DSL for modelling the workloads of projects, that has been defined in Emfatic[2] (a convenience textual syntax for Ecore). In Listing 1, the **package** keyword marks the name of the main package, that provides the logical organisational structure of the metamodel. The **attr** keyword is used to specify features with primitive types, such as strings and integers. The **val** keyword is used to define containment references, whereas the **ref** keyword defines non-containment references. The metamodel specifies that a *Project* contains a list of *tasks* and a list of *people*. A *Task* has a *name*, a list of *efforts*, a *leader* and a list of *dependencies* to other *tasks*. Each *Effort* is assigned to a *person*, and it has a number of *months*. Finally, a *Cost Centre* has a *name* and it references a list of *efforts* that are charged to it.

When modelling the *efforts* of *tasks*, we would like to make use of a textual syntax instead of a graphical syntax as modelling *efforts* graphically in the form of further nodes/arrows can cause the diagram to become crowded and hamper readability.

[2] https://eclipse.org/emfatic

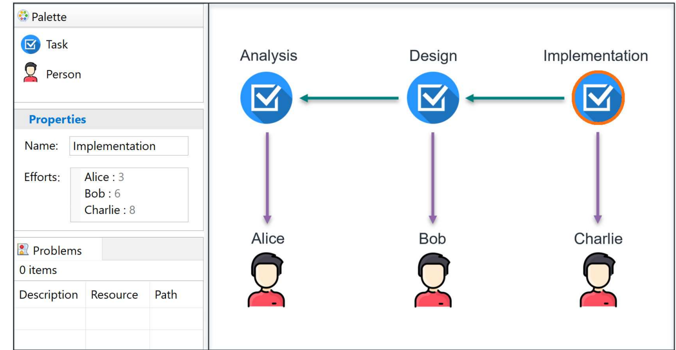

Figure 1 Hybrid graphical-textual model editor for the Project Workloads DSL

```
@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;
}

class CostCentre {
  attr String name;
  ref Effort[*] efforts;
}
```

Listing 1 Metamodel of the Project Workloads DSL

Figure 1 displays a modelled *project* in a hybrid graphical-textual model editor, in which the *tasks* and the *people* are modelled graphically, but the *efforts* are specified using 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 edges between the *tasks* represent their dependencies, i.e., the *Implementation* task is dependent on the *Design* task, and the *Design* task is dependent on the *Analysis* task. Additionally, an edge between a *task* and a *person* denotes that the *person* is the *leader* of the *task*. The *task* named *Implementation* is selected in the diagram, and its properties, i.e., the *name* and the *efforts* are displayed in the properties view. Each line from the textual representation of the *efforts* represents an *effort* model element, e.g., the second line is an *effort* that refers to a *person* named *Bob* and has a

value of 6 *months*. Although users only see and modify the textual representation of the *efforts*, behind the scenes they interact directly with the *efforts* list from the metamodel (i.e., `val Effort[*] efforts`). The textual representation of the *efforts* is parsed whenever it is modified, and it produces *effort* model elements. Then, reference resolution is carried out to resolve the references of the *efforts* to *person* model elements.

Note that this is a minimal contrived example to demonstrate the motivation and contributions of this work while keeping accidental complexity to a minimum. Cases of real-world languages with a hybrid graphical-textual syntax include 4Diac[3], CaMCOA (Cooper et al. 2021), Matlab Simulink, and various flavours of state machines that provide a graphical syntax for states and transitions and a textual syntax for guards and actions.

2.3. Requirements

In the following, we will use the motivating example to illustrate a set of required capabilities for hybrid graphical-textual model editors, that are motivated by the requirements and challenges that have been thoroughly described in (Cooper & Kolovos 2019).

2.3.1. References between Textual and Graphical Parts

For the purpose of defining complex behaviour or expressions, the textual language must be able to reference model elements that have been defined graphically in the diagram.

In Figure 1, the textual YAML-like syntax is referencing graphical model elements of type *person* from the diagram, e.g., the first line references a *person* with the name *Alice*. The model editor should support navigation from the textual editor into the diagram to the referenced graphical model elements. By performing e.g. control-click on *Alice* from within the textual editor, the model editor should navigate to the diagram definition of the graphically defined *person* named *Alice*. Similarly, users should be able to find all references to *Alice* both within diagrams and within textual expressions in an integrated manner.

2.3.2. Textual Editors with Assistance Features

The textual languages must benefit from the developer assistance features that are included in most of the modern integrated development environments (IDEs). These features include syntax highlighting, auto-completion, error detection markers, and refactoring. Including these features in the textual editors will eliminate the need for the developer to manually check the typed text, and this can result in increased productivity.

As the user is typing the *name* of a *person* in the textual editor, an auto-completion menu should be displayed, either automatically, or on demand by using keyboard shortcuts such as control+space. The auto-completion menu should display the names of all *people* that exist in the diagram. If one types the *name* of a *person* that does not exist in the diagram, the textual editor should display an error marker that informs the user about the problem.

[3] https://www.eclipse.org/4diac/en_ide.php

2.3.3. Consistency Enforcement

Considering that the textual model elements can reference the graphical model elements, they must remain consistent and synchronised. As the model evolves, the model editor must automatically enforce consistency between the textual and graphical parts of the model. For example, if a graphical model element is deleted from the diagram, then any textual model element that was referencing the graphical model element must be updated by unsetting its reference.

In a rename refactoring operation, a model element is renamed, and the new name is propagated to all model elements that were referencing the renamed model element. If the *person* named *Charlie* is renamed into *David*, in the context of a rename refactoring operation, then the textual representation of any *effort* that was referencing the *person* named *Charlie* must be updated with the new refactored name. Considering this scenario in Figure 1, the third line of the *efforts* textual representation would be updated as follows: "`David:8`". Alternatively, if the *person* named *Charlie* is renamed outside of the context of a rename refactoring operation, or if it is deleted, then any *effort* that was referencing the *person* should lose its reference to the *person*, and consequently, the *effort*'s textual representation should be updated to reflect the lost reference. Considering this scenario in Figure 1, the third line of the *efforts* textual representation would be updated as follows: "`:8`", as the textual syntax is permissive and allows unset references.

2.3.4. Uniform Error Reporting

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.

The *effort* from the first line references the *person* named *Alice*, however, if this *person* does not exist in the diagram, this would cause the reference resolution to fail. Therefore, the model editor must report this model inconsistency as an error. The user should be able to click on the error marker and be redirected to the problematic model element, i.e., the *task* named *Implementation* that contains *efforts* with unresolved references.

2.3.5. Integrated Abstract Syntax Graph (ASG)

The model that is expressed through a hybrid graphical-textual syntax must be exposed to model management programs (e.g. model-to-model or model-to-text transformations) as a single unified abstract syntax graph (ASG) that integrates elements from both the textual and graphical parts of the modelled system.

The modelled *project* from Figure 1 must be exposed to model management programs as a single ASG that integrates the textual parts of the model (i.e., the *efforts*), and the graphical parts of the model (i.e., the *tasks* and the *people*). Therefore, the textual representation of the *efforts* must not be exposed as plain text to model management programs, but rather as a list of *effort* model elements.

3. Background

This section briefly describes the key notions and technologies that are required for the proper understanding of this paper.

3.1. Textual and Graphical Modelling

Textual model editing is often realised via a background parsing strategy (Scheidgen 2008). Background parsing is a circular process that has the following steps: the user edits text using a text editor, then the text is parsed according to a grammar, and a model is derived in memory from the resulting parse-tree based on a given grammar to metamodel mapping. Finally, the in-memory representation of the derived model can be validated, to check whether it satisfies various constraints.

Graphical modelling is done by adding, removing and editing visual elements such as symbols and icons, on top of diagrams. Graphical model editors are typically based on the *Model-View-Controller* (MVC) pattern (Scheidgen 2008). MVC is a software design pattern that separates presentation and interaction from the system data (Sommerville 2010). A graphical model editor displays representations of model elements (*Model*) through view objects (*View*) (Scheidgen 2008). Users can create new model elements, edit the values of model element attributes, and delete model elements by using a set of actions. The actions that can be executed by the users are specified in controller objects (*Controller*). The editor allows users to edit model elements directly, and the controller reacts to these changes and updates the representation of the model.

Multi-view modelling makes use of textual and graphical modelling. Multi-view modelling focuses on the creation of viewpoints that are materialised through textual and/or graphical views (Addazi & Ciccozzi 2021). Views are composed of one or more models. Multi-view modelling aims at providing consistency across different views. Projectional editors often support multi-view modelling capabilities, as one can define multiple projections for an abstract syntax.

Blended modelling is similar to multi-view modelling, nevertheless, the focus is not on identifying viewpoints and views, but on providing multiple blended editing and visualising notations to interact with a set of concepts (Ciccozzi et al. 2019). With blended modelling, one can choose and switch between several different notations for the same concepts or parts of the abstract syntax. Blended modelling can be considered orthogonal to multi-view modelling, as it aims to provide a multi-notation characterisation that may be used to define viewpoints or views (Ciccozzi et al. 2019).

3.2. Modelling Technologies

In the following, we present the modelling technologies that have been used or are related to our research.

- **Eclipse Modelling Framework (EMF)**[4]. The Eclipse Modelling Framework (EMF) is the core MDE technology in Eclipse. EMF uses Ecore for metamodelling, and the XML Metadata Interchange (XMI) format for storing models. EMF has code generation facilities that can produce modelling editors from metamodels and a Java-based API that can manipulate models programmatically (Brambilla et al. 2017). One advantage of using EMF is that it provides interoperability with EMF-based modelling frameworks. Many modelling frameworks are based on EMF, such as Xtext, Sirius, EMFText, EMF.Cloud, Acceleo, ATL and VIATRA.
- **Ecore**. EMF provides the Ecore metamodelling language, which is the de facto standard for metamodelling within the academic MDE community. Ecore is a simplified implementation of Essential MOF (EMOF), and it is used in the Eclipse implementation of UML, and in many other modelling tools. Ecore-based metamodels can be defined using a graphical syntax or textual syntax. Eclipse Emfatic[5] and Xcore are textual syntaxes for Ecore metamodels.
- **Xtext**[6]. Xtext enables the definition of textual syntaxes for EMF-based modelling languages. Starting from an EBNF-based grammar specification, Xtext automatically generates the tooling for a textual language, such as a linker and a parser (Bettini 2016). From an Xtext grammar, one can derive an Ecore-based metamodel for representing the abstract syntax. Alternatively, one can import an existing Ecore-based metamodel to be used in the grammar. Xtext generates an Eclipse textual editor that includes developer assistance features, such as syntax highlighting, auto-completion, and error detection. The dependency injection facilities in Xtext can be used to customise features in the generated tooling, such as syntax highlighting, scoping and refactoring behaviour (Cooper & Kolovos 2019).
- **TEF**[7]. TEF is a textual modelling framework that can automatically generate EMF-based textual editors. TEF provides a syntax definition language called TSL, that can describe a textual notation for an Ecore metamodel. Using a TSL specification, one can automatically generate textual editors with developer assistance features, such as syntax highlighting, auto-completion and error detection.
- **Graphical Modelling Framework (GMF)**. GMF is an Eclipse framework for building graphical DSLs and their supporting editors. It is built on top of EMF and the Graphical Editing Framework (GEF) (Seehusen & Stølen 2011). GEF is a framework that enables the creation of rich graphical editors and views within Eclipse. With GMF, the abstract syntax of a language is specified using an Ecore-based metamodel, and the graphical concrete syntax is implemented using GEF (Seehusen & Stølen 2011).
- **Sirius**[8]. Sirius is a framework built on top of GMF that enables the creation of graphical DSLs and their supporting editors (Viyović et al. 2014). The abstract syntax of a language is specified using an Ecore-based metamodel, and the graphical concrete syntax is specified using a Viewpoint Specification Model (VSM). The graphical editors can be extended by building custom property views and by using custom Java services (Cooper & Kolovos 2019).

[4] https://www.eclipse.org/modeling/emf

[5] https://www.eclipse.org/emfatic

[6] https://www.eclipse.org/Xtext

[7] https://github.com/markus1978/tef

[8] https://www.eclipse.org/sirius

4. Related Work

This section presents the state of the art with regard to hybrid graphical-textual model editors and related previous work.

A technique for embedding textual modelling into graphical modelling has been presented in (Scheidgen 2008). The integration is based on TEF and GMF. TEF has been extended to be able to generate embedded textual editors that can be integrated with EMF-generated tree-based editors and graphical editors implemented using GMF. The textual editors include assistance features, such as syntax highlighting, code completion and error markers. Parts of the abstract syntax are associated with a graphical concrete syntax, and one can choose to view the textual representation of any graphical model element in a TEF textual editor. Therefore, it can be argued that one cannot have model elements that are purely textual, as one can only view the textual representation of a graphical model element. The advantage of this approach is 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. In the textual representation, one can reference graphical model elements from the diagram. When the textual representation is modified and saved, the changes are committed and merged with the underlying graphical model element. The modelled system can transparently be exposed to model management programs as a unified ASG. However, in this work, consistency between the textual and graphical parts of the model and uniform error reporting is not addressed.

Obeo[9] and Typefox[10] have presented two case studies on the integration of Xtext and Sirius (Obeo & TypeFox 2017). The first case study presents how the same EMF model can be edited either graphically or textually. Figure 2 displays the first case study, and it illustrates a use case for analysing farming activities, with the goal of optimising the water consumption used to irrigate a given exploitation. The user can choose to edit the model either graphically or textually, depending on which approach is more suitable for editing specific model elements. Model editing is performed through two different editors, the Sirius graphical editor and Xtext textual editor. As the model is modified and saved in one editor, the other editor becomes synchronised. In this case study, a graphical syntax and a textual syntax is used, independently of each other, and this approach does not align with our intent of having hybrid (part-graphical and part-textual) syntaxes.

In the second case study, which is presented in Figure 3, the user can edit the model graphically and textually from within the same model editor. An Xtext textual editor has been embedded in the Sirius graphical model editor. This case study uses a Sirius-based graphical DSL, and an Xtext-based textual DSL that references the elements from the graphical DSL. The graphical model elements have textual properties that contain textual expressions. When a graphical model element is selected in the Sirius editor, the embedded Xtext editor is displayed in the Eclipse "Properties" view. The textual expressions can be written using assistance features such as syntax highlighting

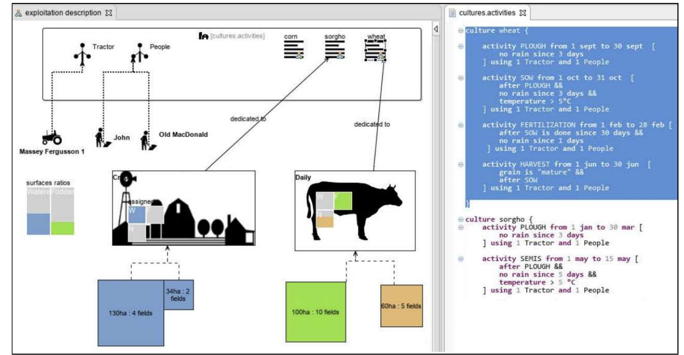

Figure 2 Xtext and Sirius editors - Farming case study (Obeo & TypeFox 2017)

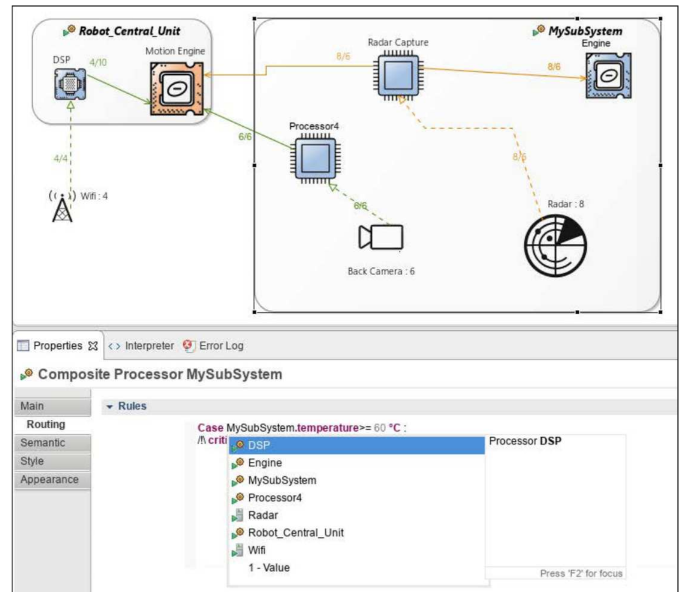

Figure 3 Hybrid graphical-textual DSL for modelling hardware components (Obeo & TypeFox 2017)

and code completion. In the embedded Xtext editor, a user can control-click a graphical model element that is referenced from a textual expression to navigate to the graphical model element in the Sirius diagram. However, the model editor does not enforce consistency between the textual and graphical parts of the model. Errors are only reported when the embedded textual editor associated with a textual expression is being displayed. Otherwise, if the embedded textual editor is no longer displayed, then the user would not be aware of whether an error exists in the textual expression. Hence, errors are not reported uniformly. Finally, model management programs only see the textual properties containing textual expressions as plain text, rather than as model elements.

Based on (Obeo & TypeFox 2017), further research has been carried out in (Cooper 2018), with the aim of embedding textual DSLs into graphical model editors. References between the textual and graphical parts of the model are supported by the model editor. For the purpose of enabling rename refactoring in the textual parts of the model, the textual expressions have

[9] https://www.obeosoft.com
[10] https://www.typefox.io

Table 1 Requirements fulfilment of state-of-the-art hybrid graphical-textual model editors

Reference	References between textual and graphical parts	Textual Editors with Assistance Features	Consistency Enforcement	Uniform Error Reporting	Integrated ASG
(Scheidgen 2008)	✓	✓	✗	✗	✓
(Obeo & TypeFox 2017)	✓	✓	✗	✗	✗
(Cooper 2018)	✓	✓	✗	✓	✗
(Altran 2022)	✓	✓	✗	✗	✓

been stored in separate files on the file system and links to the files have been stored in the model. By using this approach, one can benefit from Xtext's default refactoring engine that relies on files, and no further extensions are required to enable rename refactoring for the textual expressions. Therefore, this work ensures the synchronisation between the textual and graphical parts of the model in the event of a rename refactoring operation. Except for the synchronisation in the event of a rename refactoring operation, the model editor does not enforce consistency between the textual and graphical parts of the model. Uniform error reporting has been realised by implementing a custom builder that is executed each time the project is built, parsing all textual expressions to identify the potential errors and report them in the Eclipse "Problems" view. In this work, no mechanism is provided to transparently expose the textual expressions as model elements to model management programs.

Altran, which is known now as Capgemini Engineering[11], has extended the work from (Obeo & TypeFox 2017) by embedding Xtext textual editors into Sirius graphical diagrams, in addition to the Eclipse "Properties" view (Altran 2022). References between the textual and graphical parts of the model are supported, however the model editor does not enforce consistency between the textual and graphical parts of the model. The model editor does not report errors in the Eclipse "Problems" view, as errors are reported only in a pop-up window whenever the user types syntactically incorrect textual expressions. The textual editors that are embedded in the diagrams display the textual representation of the underlying model element. Whenever the user modifies the textual representation in a textual editor that is embedded in the diagram, the textual representation is parsed and the resulting model element is merged with the underlying model element. Therefore, the textual expressions can transparently be exposed as model elements to model management programs.

An add-on extension has been developed for Capella in (EclipseFoundation 2022) that integrates an Xtext textual editor with assistance features in a graphical model editor. Capella[12] is a Sirius-based tool for modelling complex and safety-critical systems in embedded systems development for industries such as aerospace, transportation, space and automotive. Similarly to the first case study from (Obeo & TypeFox 2017), one can choose to edit the model either graphically or textually, depending on which approach is more appropriate. Synchronisation between the textual and graphical editor is provided, as a change from the textual editor is propagated to the graphical editor and vice versa. A graphical syntax and a textual syntax is used independently of each other, and this approach is in contrast with our intent of having a hybrid graphical-textual syntax.

The work from (Addazi & Ciccozzi 2021) mixes graphical and textual modelling, as it relies on blended modelling. In (Addazi & Ciccozzi 2021), textual editors with assistance features are used, and one can reference graphical model elements from the textual syntax. However, uniform error reporting is not addressed in this work. Our hybrid graphical-textual modelling approach differs from the one of blended modelling. Blended modelling focuses on providing several different notations, which can be graphical or textual, for the same concepts of the abstract syntax, and it is concerned with keeping the different notations of the same part of the abstract syntax synchronised. However, our work focuses on making use of graphical and textual notations for different concepts of the abstract syntax, and it is concerned with maintaining the consistency of the references between the graphical and textual parts of the model.

We conclude that there are four main works that follow the same line of work as ours. The works from (Scheidgen 2008; Obeo & TypeFox 2017; Cooper 2018; Altran 2022) focus on model editors that meet several of the requirements presented in Section 2.3. Table 1 presents the requirements that are fulfilled by each of these works.

5. Graphite

We have developed Graphite[13], a tool for streamlining the development of hybrid graphical-textual Sirius/Xtext-based model editors, that meet the requirements discussed in Section 2.3. The tool consists of a set of reusable services and model transformations that contain approximately 2000 lines of code. This section discusses our contributions, that have been developed into Graphite.

5.1. Architecture and Usage

This section presents how Graphite can be leveraged to develop a hybrid graphical-textual model editor, as the one from Figure 1. The components and the steps involved in developing the model editor are illustrated in Figure 4.

Figure 4 illustrates that the first step is the definition of the abstract syntax and of the individual concrete syntaxes. Initially,

[11] https://capgemini-engineering.com
[12] https://www.eclipse.org/capella
[13] https://github.com/epsilonlabs/graphite

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

Listing 2 Modified metaclass with added textual property and annotation

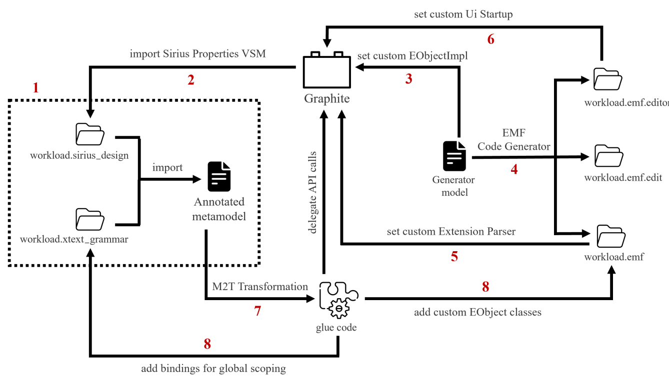

Figure 4 Graphite architecture

one must define the abstract syntax, i.e., the metamodel of the Project Workloads DSL. In the case of the motivating example, the metamodel from Listing 1 has been defined. However, if one would like to use Graphite, then the metamodel must be modified. For each property from the metamodel that one would like to express with a textual syntax, a new textual property must be added in the metamodel. Modifying the metamodel by adding a new textual property is not ideal, however, this design decision has been made to be able to tolerate temporary inconsistencies in the model editor. Section 5.2 presents how temporary inconsistencies are tolerated by having a textual property that stores the textual representation. Additionally, an annotation must be added to the metamodel, to define the mapping between the added textual property and the underlying model element.

To express the *efforts* with a textual syntax, one must modify the *Task* metaclass as in Listing 2. Listing 2 specifies that the textual representation of the *efforts* list is stored in the textual property *effortsExpression*. The annotation specifies the grammar that is used to parse the *efforts* textual expression, and additionally, it specifies which is the property expressed with a textual syntax (i.e., the *efforts* list). Note that in the motivating example, we would like to express only the *efforts* property with a textual syntax, therefore only one textual property and one annotation has been added to the metamodel. However, if one wanted to express multiple properties from the metamodel using a textual syntax, then a textual property and an annotation would have been added to the metamodel for each of them.

With Graphite, the textual syntax must be specified using Xtext grammars. For the motivating example, one must define an Xtext grammar for a YAML-like textual syntax. Listing 3 presents the grammar that is used for parsing the textual representation of the *efforts*, which derives a list of *effort* model elements. The grammar must import the metamodel to specify which parts of the abstract syntax are expressed with a textual syntax. The grammar from Listing 3 specifies that whenever the textual representation of the *efforts* list is parsed, a *task* that contains a list of *effort* model elements is derived. To enable references between the textual and graphical parts of the model, the grammar must reference model elements that are expressed with a graphical syntax. Line 13 from Listing 3 specifies that the *efforts* textual expression can reference a model element of type *Person* that is modelled graphically.

```
grammar workload.xtext_grammar.Efforts
    with org.eclipse.xtext.common.Terminals

import "ProjectWorkloadsDSL"
import "http://www.eclipse.org/emf/2002/Ecore"

Main returns Task:
  {Task}
  (efforts+=Effort (NEWLINE efforts+=Effort)*)?;

Effort returns Effort:
  {Effort}
  (person=[Person])? ':' months=INT;

terminal NEWLINE:
  (' '|'\t')* '\r'? '\n' (' '|'\t')*;
```

Listing 3 *Efforts* Xtext grammar for a YAML-like textual syntax

A Sirius Viewpoint Specification Model (VSM) must also be defined to map parts from the abstract syntax to a graphical syntax. In the motivating example, we would like to model graphically the *tasks* and the *people*. In a Sirius VSM, one must map model elements of type *Task*, and model elements of type *Person* to various graphical symbols. After the graphical syntax has been defined, the first step from Figure 4 has been finalised.

In the second step, one must import into the defined Sirius VSM a custom Sirius VSM provided by Graphite. The custom Sirius VSM from Graphite overwrites the default properties view from Sirius, such that for each property that is expressed with a textual syntax, as defined in the annotated metamodel, an Xtext textual editor is embedded in the properties view of the model editor. To realise this behaviour, one must additionally extend the Java Services class of Sirius with a custom class from Graphite, that provides services that are required by the custom Sirius VSM. Note that the Java Services class of Sirius can be used to define a set of services for enriching the capabilities of Sirius graphical model editors.

As the third step, one must change the default EMF EObject type in the EMF generator model to a custom EObject imple-

mentation from Graphite. When using EMF-based editors, EMF attaches to model elements an ordered set of default event listeners (i.e., EMF Adapter objects) that trigger certain actions to be invoked in response to various events. For addressing consistency enforcement in Section 5.7, custom event listeners have been attached to referenced model elements. For a specific event, multiple event listeners may be triggered, thus the order in which event listeners are attached to model elements is important, as invoked actions by different event listeners may be in conflict with each other. To avoid such conflicts, a custom EObject type was implemented, to specify that the event listeners which ensure consistency enforcement are the last event listeners to be triggered.

In the following step, the EMF code generator must be triggered to generate the projects for the model code, the edit code and the editor code, as illustrated in Figure 4.

For the fifth and sixth steps, one must configure extension points to Graphite. In the model project (i.e., "workload.emf"), an extension point must be configured to an Extension Parser (i.e., "org.eclipse.emf.ecore.extension_parser"). The Extension Parser maps a specific model resource extension to a custom resource factory from Graphite, that instantiates resources with custom behaviour. In Graphite, a custom resource type has been implemented, that overwrites the default *onLoad* and *onSave* events listeners of a resource. The *onLoad* event listener has been overwritten to parse all textual expressions when the resource is loaded. Additionally, the *onSave* event listener has been overwritten to specify which derived model elements must be persisted on disk, as described in Section 5.5. The next step is to configure an extension point (i.e., "org.eclipse.ui.startup") in the model editor project (i.e., "workload.emf.editor"), to a custom Startup class from Graphite that registers the model validator used for error reporting.

Finally, one must execute a model-to-text (M2T) transformation, which is also part of Graphite, that takes as input the annotated metamodel and generates glue code that delegates API calls to Graphite. The generated code is added to the model project (i.e., "workload.emf"), and to the grammar project (i.e., "workload.xtext_grammar"). A set of classes are added to the model project, that provide facilities for expressing properties with a textual syntax. Additionally, the code added to the Xtext grammar provides facilities for global scoping, that enable references between the graphical model elements defined in the Sirius diagram and the textual model elements defined in the embedded Xtext textual editors.

With Graphite, one can develop such model editors for any DSL, by defining an annotated metamodel, one or more Xtext grammars, a Sirius VSM, and a minimal configuration of extension points. Note that the developed hybrid graphical-textual model editors fulfil all requirements presented in Section 2.3.

5.2. Tolerating Temporary Inconsistencies

This section discusses the decision to use additional string attributes to record textual expressions, the alternatives considered, and their strengths and weaknesses.

To express a property from the metamodel with a textual syntax, one option is to project its textual representation whenever it is accessed. When the textual representation is modified, it is parsed and the result is merged with the underlying model element. If a user types a syntactically incorrect textual expression, parsing fails and model elements cannot be derived from it, which should have been merged with the underlying model element. Therefore, the textual model editor will report an error and ask the user to write a valid textual expression. Alternatively, the textual model editor may ask the user to confirm whether they agree to discard the syntactically incorrect textual expression and revert to the previous textual expression. The shortcoming of this approach is that it makes textual model editing unnecessarily restrictive, as temporary inconsistencies are not tolerated.

To be able to tolerate temporary inconsistencies in our model editors, a textual property has been added to the metamodel to store the textual representation of the underlying model element. Therefore, a textual property (i.e., `attr String effortsExpression`) has been added to the *Task* metaclass in the metamodel from Listing 1, to store the *efforts* in a textual format, as presented in Listing 2. The disadvantage of this technique is that textual expressions are stored in the model, therefore depending on the complexity and number of textual expressions stored, the file size of the model can increase substantially. The textual editors display the content of the textual property that has been added to the metamodel. The textual property represents a textual projection of the underlying model element, i.e., the *effortsExpression* string is a projection of the *efforts* list.

We will present an example to illustrate how temporary inconsistencies are tolerated by having a textual property that stores the textual representation of the underlying model element. The initial value of the *efforts* textual expression is "`Michael:18`", as the list of *efforts* contains only one *effort* that references a *person* named *Michael* and has a value of 18 *months*. In the textual editor, the user changes the value of the *months* to 36 and then accidentally enters a typo, e.g., a dot between the digits of the number. At this point, the textual editor and the textual expression have the following content: "`Michael:3.6`". However, the textual expression is syntactically incorrect, due to the fact that the *months* must be an integer, and the user has specified a float (i.e., the float 3.6). The *efforts* textual expression cannot be parsed successfully and an *effort* model element could not be derived. Currently, there is a state of temporary inconsistency, as the textual expression from the textual editor does not represent an equivalent projection of the underlying model element. The syntactically incorrect textual expression indicates 3.6 number of *months*, however, the *effort* model element from the *efforts* list has a value of 18 *months*. To conclude, the model editor tolerates temporary inconsistencies, by adding to the metamodel additional string attributes to store textual expressions.

The textual representation and the underlying model element must remain synchronised. A change in the textual representation must be reflected in the underlying model element and vice versa. Therefore, the textual property that stores the textual expression and the underlying model element must be bidirectionally synchronised.

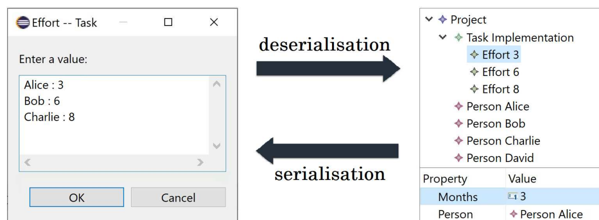

Figure 5 Bidirectional synchronisation via serialisation and deserialisation

5.3. Bidirectional Synchronisation between the Textual Representation and the Underlying Model Element

Each time the *efforts* textual expression changes, its content is parsed and model elements of type *effort* are derived and then assigned to the *efforts* list. To realise this behaviour, we have not used any out-of-the-box solution, however, we rely on Xtext APIs that provide parsing facilities to derive model elements. To facilitate the discussion from this section, we will call the *efforts* list a derived property (i.e., `val Effort[*] efforts`). It is called a derived property for the reason that it contains the model elements that are derived from parsing the textual property. The derived property represents the underlying model element(s) that is expressed through a textual syntax, and its content is an equivalent projection of the textual property.

The bidirectional synchronisation between the textual representation and the underlying model element is achieved via serialisation and deserialisation, as presented in Figure 5. Whenever the content of the derived property is modified, by e.g., adding or removing *effort* model elements or by modifying the properties or order of existing *effort* model elements, the content of the derived property is serialised as a textual expression that overwrites the value of the textual property. For this purpose, an event listener is attached to the derived property to trigger the serialisation of its content whenever it is modified. The event listener is implemented in Graphite and the glue code generated in the seventh step from Figure 4 is responsible for attaching the event listener to derived properties. The serialisation is triggered whenever model elements are added into or deleted from the derived property. Additionally, the serialisation is triggered whenever the value of any property of an existing model element stored in the derived property changes. By following this approach, the textual expression that is stored in the textual property is bidirectionally synchronised with the underlying model element, i.e., the derived property.

```
Alice:3
Bob:6
Charlie:8
```

Listing 4 Initial *efforts* textual expression

In the following, we will consider that the *effortsExpression* property has the value from Listing 4. Three *effort* model elements are derived by parsing the textual expression. Each line from Listing 4 represents an *effort*, e.g., the second line represents an *effort* that refers to a *person* named *Bob* and has a value of 6 *months*. If one changes the *months* value of the second *effort* model element to 25, then the textual expression will be updated as in Listing 5.

```
Alice:3
Bob:25
Charlie:8
```

Listing 5 Updated *efforts* textual expression after changing the *months* value

Whenever an *effort* model element is removed from the derived property, e.g., the third *effort* is deleted from the list of *efforts*, then the value of the textual property would be the same as Listing 4, excluding the third line. Correspondingly, whenever an *effort* model element is added to the derived property, e.g., an *effort* that has a value of 5 *months* and references a *person* named *David*, then the textual property would be updated as in Listing 6.

```
Alice:3
Bob:6
Charlie:8
David:5
```

Listing 6 Updated *efforts* textual expression after adding a new *effort* model element

5.4. Managing Invalid Textual Expressions

A textual expression is considered to be in an invalid state whenever it cannot be parsed successfully or whenever reference resolution fails.

We will consider an example in which a textual expression is initially syntactically correct. The derived property contains one or more model elements, that represent an equivalent projection of the textual expression. The user accidentally enters a typo in the textual expression, and the textual expression becomes syntactically incorrect. The textual expression cannot be parsed successfully, and incomplete model elements or no model elements are derived. In this case, to avoid losing the content of the derived property, we do not overwrite its content. The content of the derived property remains the same as when the textual expression was syntactically correct. Therefore, the content of the derived property represents a projection of the last valid textual expression. Note that at this point, the model editor is in a state of temporary inconsistency.

5.5. Storage of Derived Model Elements

When the model is loaded by e.g. opening it in a model editor or accessing it in model management programs, all textual properties that contain textual expressions are parsed according to their associated grammars, and for each of them, the derived model elements are assigned to the respective derived property. For example, the model elements derived by parsing the *effortsExpression* property of each *task* element are assigned to the *efforts* list of the *task*.

One may argue that there is little benefit in storing the derived model elements on disk (i.e., the *efforts* list), as one can retrieve them at runtime by parsing the textual expressions. Therefore, we considered labeling the derived model elements as transient, to not store them on disk, as we can produce them at runtime. However, this becomes problematic in a state of temporary inconsistency, when a textual expression is in an

invalid state. For example, the *efforts* textual expression has the initial content of "`Michael:18`". The *efforts* list contains one *effort* that references a *person* named *Michael* and has a value of 18 *months*. Additionally, a *cost centre* is referencing the *effort* model element from the *efforts* list. In the textual editor, the user accidentally enters a typo, e.g., a dot between the digits of the number. At this point, the textual expression is "`Michael:1.8`". As the textual expression is syntactically incorrect, it cannot be parsed successfully. This is a state of temporary inconsistency in which the textual representation of the *efforts* list does not represent an equivalent projection. In the case that the model resource is closed and saved, and the *efforts* list is not serialised on disk, then the *efforts* list cannot be recovered. When the model resource is loaded again, the syntactically incorrect textual expression cannot be parsed successfully, therefore *effort* model elements cannot be derived. Consequently, the list of *efforts* is lost. Furthermore, the *cost centre* that was referencing the *effort* model element loses its reference to the *effort*.

One mechanism to tackle this issue is to store on disk the derived model elements when the model resource is saved, but only in the case that the textual expression is in an invalid state, i.e., it cannot be parsed successfully or reference resolution fails. Hence, the derived model elements are stored on disk if the textual property is in an invalid state, but if the textual property 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. To realise this behaviour, we customised the event that is triggered when the model is saved, to store on disk only the derived model elements that represent the projection of the textual expressions that are in an invalid state. For example, if the textual representation of the *efforts* list from the *task* named *Implementation* is in an invalid state, then we store the *efforts* list on disk. However, if the textual representation of the *efforts* list from the *task* named *Design* is in a valid state, then we do not store the *efforts* list on disk, as we can recover it when the model is loaded.

5.6. Managing References to Derived Model Elements

Whenever the value of the textual property changes, it is parsed and the derived model elements must be assigned to the derived property. One option we considered was to overwrite the derived property each time with newly derived model elements. The limitation of this approach occurs whenever another model element holds a reference to a derived model element that is stored in the derived property. As new model elements (i.e., a list of *effort* model elements) are derived each time the textual property is parsed, a consistency issue occurs due to the direct assignment of the list of derived model elements to the derived property. If another model element was referencing any of the previous model elements stored in the derived property, whenever the value of the textual property changes, new model elements would overwrite the content of the derived property, and the referring model element would lose the reference to the previous model element. For example, a *cost centre* is referencing the first *effort* from Figure 1. If the user modifies the textual representation of the third *effort* from Figure 1, then the textual representation is parsed, and new *effort* model elements are derived, that will overwrite the content of the *efforts* list. Although the textual representation of the first *effort* is the same as before, in memory a new object has been produced and assigned to the *efforts* list. As the previous *effort* model element does not exist anymore in the model, the *cost centre* loses its reference to the previous *effort*.

We tackled this issue by not overwriting the derived property each time with new derived model elements, and instead applying a merging operation before assignment. For instance, if the textual property has the value as in Listing 4, the derived property would contain three *effort* model elements. When a fourth *effort* is defined in the textual property, as in Listing 6, the textual property is then parsed and four new *effort* model elements are derived. These four new *effort* model elements must be merged with the previous three *effort* model elements, and the resulting merged *efforts* are assigned to the derived property. In this example, the first three initial *effort* model elements are kept, and the three new *effort* model elements are discarded. Note that the merging operation is also applied for textual expressions that derive a single model element by parsing, rather than a list of model elements. The merging is performed by matching the identifier of objects. By default, the name property represents the identifier of a model element, however, we are able to customise this behaviour by using a data structure that maps a specific model element type to an identifier property. The data structure can be defined with minimal hand-written code in the glue code that is generated in the seventh step from Figure 4. When matching is performed, the data structure is queried to retrieve the identifier property of a specific model element type. Whenever there is a match between an old derived model element and a new derived model element, the old derived model element is kept and all its properties are overwritten with the values of the new derived model element, and then the new derived model element is discarded. Any of the old derived model elements that are not matched in the list of new derived model elements are discarded. Additionally, any of the new derived model elements that are not matched in the list of old derived model elements are kept as is, without performing any merging. Note that the order of model elements is preserved when using this merging technique. By applying this merging technique, the model elements that are referencing derived model elements do not lose their reference whenever the textual property is parsed.

5.7. Consistency Enforcement

This section discusses various challenges and solutions for consistency enforcement.

Each time a model element that is referenced by a textual expression changes its identifier, then the content of the derived property must be serialised to update the content of the textual expression. With regard to this, the challenge is to trigger the serialisation in an efficient way. A naive approach would be to trigger the serialisation whenever any property from the model changes its value, and if the new textual expression is different than the previous value of the textual property, overwrite it.

In our work, we applied an efficient technique that attaches an

event listener to each referenced model element, which triggers the serialisation of the content of the derived property only when the identifier of the referenced model element has changed. This efficient technique is applied at the instance level. The event listener is provided by Graphite, as each time a textual expression is parsed, Graphite identifies the references from the textual expression, and attaches the event listener to each reference. Each event listener is aware of all model elements that are referring to the referenced model element. In the example from Figure 5, after parsing the *efforts* textual expression, the referenced *person* model elements are identified, and an event listener is attached to each referenced *person*. In this case, an event listener is attached to the *persons* named *Alice*, *Bob* and *Charlie*, whereas to the *person* named *David*, no event listener is attached. For example, when the name of the *person Alice* is changed, then the serialisation of the content of the derived property is triggered, and the textual expression is updated.

If the change of the identifier's value is performed outside the context of a rename refactoring operation, then all model elements that are referring to the referenced model element lose their reference, and then serialisation is performed. In the case of a rename refactoring operation, the references are kept, and only serialisation is performed. However, this efficient technique does not work when reference resolution fails, as the *person* model elements cannot be identified.

5.8. Uniform and Efficient Error Reporting

To realise uniform and efficient error reporting, our solution is to store in memory the diagnostics information, i.e., the errors that are produced when parsing a syntactically incorrect textual expression and when reference resolution fails. As the textual expressions are parsed only when necessary, the efficiency of this solution is ensured.

When a validation operation is triggered in the model, then the diagnostics information that is stored in memory is used to populate error markers in the problems view.

This solution does not work in the case that the textual expression is in an invalid state. For example, the *efforts* textual expression has been initially parsed successfully, and no errors have been yielded. A *person* is then deleted, which was previously referenced by the textual expression. At this point, the textual expression is in an invalid state, as it references a *person* that does not exist. As the reference resolution operation has not been triggered, the model editor is not aware that the textual expression is in an invalid state. Therefore, no errors are reported in the problems view, as the last time the textual expression was parsed, no errors have been yielded.

The solution to store in memory the diagnostics information only works when used in conjunction with the consistency enforcement techniques that have been applied in Section 5.7. By applying the consistency enforcement techniques, in the context of the previous example, when the referenced *person* is deleted, then the textual expression is updated to reflect the lost reference, and now the textual expression is in a valid state, therefore no errors must be reported.

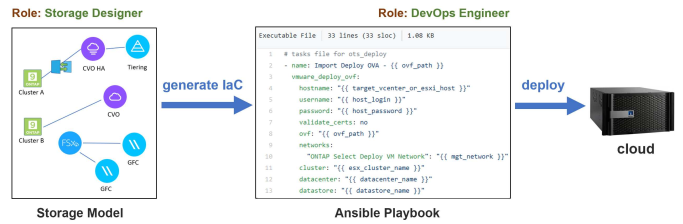

Figure 6 Case study - code generation

6. Evaluation

This section evaluates Graphite by means of an industrial case study provided by NetApp.

6.1. Case Study

NetApp[14] is a global software company that delivers hybrid cloud data services and data management services. Within NetApp, infrastructure automation is often realised via Infrastructure as Code (e.g., Ansible).

Infrastructure as Code (IaC) is a DevOps practice that enables infrastructure automation using software development practices. The management and provisioning of infrastructure is performed through definition files that contain declarative code. Once the infrastructure is defined using code, it is rolled out to systems through automated processes (Morris 2016). Ansible[15] is one example of infrastructure automation technology that delivers IaC. Ansible Playbooks are the core component of Ansible, and use the human-readable syntax of YAML.

When planning enterprise data storage environments, typically several parties and roles are involved, e.g., cloud architects, storage architects, security experts, operational teams and legal departments. Thereby, non-technical professionals often have a lack of shared understanding of the described infrastructure. For this industry scenario, we have developed a dedicated hybrid graphical-textual DSL for modelling NetApp Public Cloud Services (PCS), to lower the entry barrier to cloud services adoption and to support a shared understanding of the involved parties. The DSL uses a graphical syntax for simplification of high-level infrastructure components, and textual syntaxes for defining low-level details of NetApp PCS.

Figure 6 illustrates how the DSL simplifies cloud automation. A storage designer models an infrastructure environment by collaborating with the previously mentioned parties. Then, an equivalent Ansible Playbook of the modelled infrastructure is automatically derived. A DevOps engineer may further elaborate on the derived Ansible Playbook for fine-tuning or for including sensitive credentials. Finally, the Ansible Playbook is executed, and all infrastructure components that have been defined in the model editor are deployed to the cloud.

An excerpt from the metamodel of the DSL is presented in Figure 7, which contains the main concepts of the domain.

Open Network Technology for Appliance Products (ONTAP) is NetApp's proprietary operating system that offers many storage efficiencies, and it is deployed on physical or virtual ap-

[14] https://www.netapp.com
[15] https://www.ansible.com

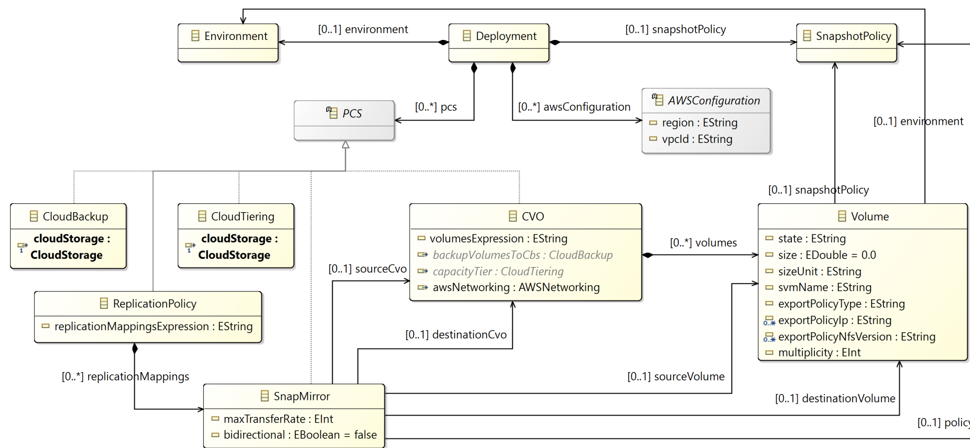

Figure 7 Excerpt of the NetApp Infrastructures metamodel

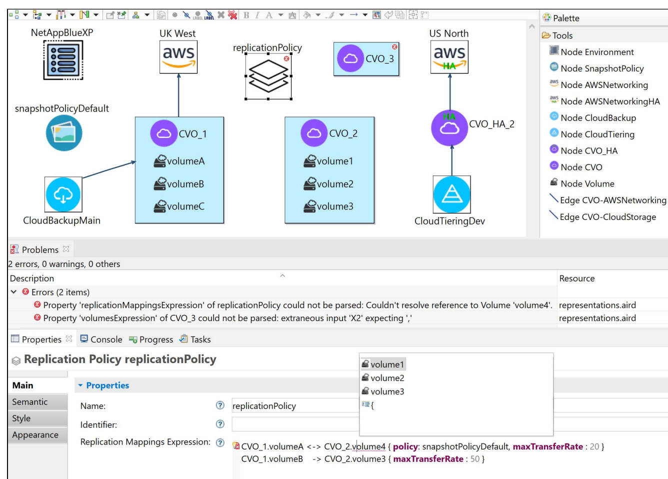

Figure 8 Infrastructures Hybrid Graphical-Textual Model Editor

pliances. Cloud Volumes ONTAP[16] (CVO) is an instance of the ONTAP operating system that is deployed to the cloud. A *Deployment* configuration contains a *SnapshotPolicy* and a list of NetApp infrastructure PCS that will be deployed to a specific *Environment* (e.g., NetApp BlueXP[17]), in a specific public cloud (AWS is currently supported). The list of NetApp infrastructure PCS can contain instances of *CloudBackup*, *CloudTiering*, *ReplicationPolicy*, *SnapMirror* and *CVO*. A *CVO* instance contains a list of *volumes* that define logical storage areas. *SnapMirror* is a proprietary protocol for replicating data from a source *volume* to a target *volume* of a *CVO* instance. A *SnapMirror* instance takes a *SnapshotPolicy* and *maximum transfer rates* (measured in mebibyte per second) into account. A *ReplicationPolicy* contains a list of *replication mappings*, i.e., multiple *SnapMirror* instances.

Figure 8 presents the hybrid graphical-textual model editor that has been developed for the DSL, by leveraging Graphite. The majority of the NetApp infrastructure PCS are represented with a graphical syntax, i.e., *CVO*, *SnapshotPolicy*, *CloudBackup*, *CloudTiering*, *AWSNetworking* and *Environment*. However, the *replication mappings* of a *ReplicationPolicy* are better suited to be represented using textual expressions. In Figure 8, the textual editor of the property *Replication Mappings Expression* is an embedded Xtext editor with assistance features. The first line specifies a bidirectional replication, whereas the second line specifies a unidirectional replication. The second line states that *volumeB* from *CVO_1* must be replicated into *volume3* from *CVO_2* with a *maximum transfer rate* of 50 mebibytes per second. Additionally, a JSON-like textual syntax was used to define the *volumes* of *CVO* instances. In this case, the *volumes* can be defined graphically, but also textually. The advantage of defining the *volumes* with a textual JSON-like syntax is that engineers can reuse JSON files containing data of *volumes* from NetApp internal systems. As the *replication mappings* and the *volumes* are expressed with a textual syntax, a textual property and an annotation has been added for each in the metamodel.

6.2. Discussion

In this section, we evaluate the model editor that has been developed for the case study by using Graphite.

In Figure 8, the textual expression references multiple graphical model elements. It references *CVO_1*, *CVO_2*, *volumeA*, *volumeB*, *volume3*, and *snapshotPolicyDefault*. By performing control-click over *snapshotPolicyDefault* from within the textual editor, the model editor navigates to the *snapshotPolicyDefault* graphical model element in the diagram.

The textual editor for the *replication mappings* textual expression has developer assistance features. It includes syntax highlighting, as the properties *policy* and *maxTransferRate* are highlighted in the textual expression. When typing the name of a *volume*, an auto-completion menu is displayed in Figure 8, which lists all *volumes* from *CVO_2*. The reference to *volume4* cannot be resolved, as it is not defined in the diagram. Accordingly, an error detection marker is displayed in the textual editor. After replacing *volume4* with *volume1* in the textual expression, reference resolution is carried out successfully. Therefore, the error marker from the textual editor disappears.

[16] https://cloud.netapp.com/ontap-cloud
[17] https://www.netapp.com/bluexp

```
CVO_1.volumeA  <->  CVO_2.volume1 { policy: snapshotPolicyDefault, maxTransferRate : 20 }
CVO_1.volumeB   ->  CVO_2.volume3 { maxTransferRate : 50 }
```

Listing 7 Textual representation of the *replication mappings* list

```
- source_cvo_name: CVO_1
  destination_cvo_name: CVO_2
  source_volume_name: volumeA
  destination_volume_name: volume1
  max_transfer_rate: 20
  policy: snapshotPolicyDefault

- source_cvo_name: CVO_2
  destination_cvo_name: CVO_1
  source_volume_name: volume1
  destination_volume_name: volumeA
  max_transfer_rate: 20
  policy: snapshotPolicyDefault

- source_cvo_name: CVO_1
  destination_cvo_name: CVO_2
  source_volume_name: volumeB
  destination_volume_name: volume3
  max_transfer_rate: 50
```

Listing 8 Generated Ansible Playbook from the *replication mappings* list

When *volumeB* is renamed in the diagram into *volumeD*, then the textual expression is updated by replacing *volumeB* with *volumeD*. Additionally, when *snapshotPolicyDefault* is removed from the diagram, the first line of the textual expression is updated, such that the *policy* property is removed from the curly braces, as it is not referring to any model element.

In Figure 8, there are two reported errors. As the *replication mappings* textual expression is referencing a *volume* that does not exist, an error is reported in the Eclipse “Problems” view. The textual representation of the *volumes* from *CVO_3* is syntactically incorrect, therefore an additional error is reported in the Eclipse “Problems” view. When the user clicks on the first error marker, the model editor redirects the user to the problematic model element, i.e., the *replicationPolicy* that has a textual expression with an unresolved reference to *volume4*.

The modelled infrastructure environment is exposed to model management programs as a single ASG that integrates the textual and the graphical parts of the model. A model-to-model transformation was used to convert the list of *replication mappings* into an Ansible Playbook. The Epsilon Transformation Language (ETL)[18] and the EMC YAML driver[19] have been used to transform the EMF model into an YAML model. The YAML model represents the generated Ansible Playbook. For every *replication mapping* that is not bidirectional (->), a YAML mapping node is added to the Ansible Playbook, that contains all properties of the *replication mapping*. However, if a *replication mapping* is bidirectional (<->), two YAML mapping nodes are added to the Ansible Playbook, that contain the same properties, with the exception that the source *CVO* and source *volume* are swapped with the destination *CVO* and destination *volume*.

[18] https://www.eclipse.org/epsilon/doc/etl

[19] https://www.eclipse.org/epsilon/doc/articles/yaml-emc

Listing 7 contains the textual representation of the list of *replication mappings*. As the first *replication mapping* is bidirectional, it is added twice to the Ansible Playbook, as in Listing 8.

To conclude, with only four hand-written lines of code, two grammars, and minimal configurations, we were able to leverage Graphite to develop a model editor that fulfils all requirements presented in Section 2.3.

7. Limitations

The merging operation of model elements is performed by matching their identifiers. One can specify the identifier of a model element type, however, no mechanism is provided to enforce model elements with unique identifiers, therefore conflicts and inconsistencies can result during the merging operation.

When the model is loaded, all textual expressions are parsed and the derived model elements are assigned to their associated derived property. There are cases in which one textual expression refers to the elements of another textual expression. To exemplify, the derived model elements from a textual expression may reference the derived model elements from another textual expression. In such cases, the order in which the textual expressions are parsed is important. References cannot be resolved if a primary textual expression is parsed before a secondary textual expression that is referenced by the primary textual expression. Our contribution does not currently address this limitation, however, it is useful in its current state, for textual expressions that do not refer to each other.

It is essential that a grammar used for parsing a textual property contains a grammar rule which populates all properties of the derived model element(s). We will consider the case in which a grammar only sets the value of *person* for each *effort*, without setting the value of the *months* property. The textual property would be parsed, and it would derive *effort* model elements that have no value for *months*. If one changes the value of an *effort*’s *months* in a model editor, then whenever new *efforts* are derived that will overwrite the old *efforts*, the *months* value that was set in the model editor would be lost. To avoid this issue, one could validate that the grammar contains a grammar rule that sets all properties of a derived model element (i.e., for each *effort* it must set the *person* and *months*).

8. Conclusions and Future Work

In this paper, we motivated a set of requirements for hybrid graphical-textual model editors and addressed several open challenges. We presented our main contribution, Graphite, which is a tool for streamlining the development of hybrid graphical-textual model editors, by using as little hand-written code as possible. We then evaluated Graphite in an industrial case study.

In future work, we will conduct experimental evaluations related to performance aspects, as the evaluation of Graphite is currently limited to a case study. We plan to address the limitations of our solution by enforcing model elements with unique identifiers, enabling textual expressions to reference other textual expressions, and validating that grammars contain a grammar rule that sets all properties of a derived model element. Additionally, we would like to extend the capabilities of the model editors, by providing facilities for finding all references of a model element, both in diagrams and textual expressions. Furthermore, we would like to investigate consistency enforcement techniques in the context of model elements that must satisfy various constraints.

Acknowledgments

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

References

Addazi, L., & Ciccozzi, F. (2021). Blended graphical and textual modelling for UML profiles: A proof-of-concept implementation and experiment. *Journal of Systems and Software*, *175*, 110912.

Altran. (2022). Xtext Sirius integration [Computer software manual]. ([Online]. Available: https://altran-mde.github.io/xtext-sirius-integration.io)

Bettini, L. (2016). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing.

Brambilla, M., Cabot, J., Wimmer, M., & Baresi, L. (2017). *Model-Driven Software Engineering in Practice* (Second ed.). Morgan & Claypool Publishers.

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

Cooper, J. (2018). *A Framework to Embed Textual Domain Specific Languages in Graphical Model Editors* (Unpublished master's thesis). University of York.

Cooper, J., De la Vega, A., Paige, R., Kolovos, D., Bennett, M., Brown, C., ... Rodriguez, H. H. (2021). 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)* (p. 308-319).

Cooper, J., & Kolovos, D. (2019). 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)* (pp. 322–325).

EclipseFoundation. (2022). Capella Textual Editor Extension [Computer software manual]. ([Online]. Available: https://github.com/eclipse/capella-textual-editor)

Morris, K. (2016). *Infrastructure as Code: Managing Servers in the Cloud*. "O'Reilly Media".

Obeo, & TypeFox. (2017). Xtext Sirius integration - white paper [Computer software manual]. ([Online]. Available: https://www.obeodesigner.com/resource/white-paper/WhitePaper_XtextSirius_EN.pdf)

Scheidgen, M. (2008). Textual Modelling Embedded into Graphical Modelling. In *European Conference on Model Driven Architecture-Foundations and Applications* (pp. 153–168).

Seehusen, F., & Stølen, K. (2011). An Evaluation of the Graphical Modeling Framework (GMF) Based on the Development of the CORAS Tool. In *International Conference on Theory and Practice of Model Transformations* (pp. 152–166).

Sommerville, I. (2010). *Software Engineering* (Ninth ed.). Pearson.

Viyović, V., Maksimović, M., & Perisić, B. (2014). Sirius: A rapid development of DSM graphical editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014* (pp. 233–238).

About the authors

Ionut Predoaia is a PhD candidate and Research Associate in the Department of Computer Science at the University of York. He is also an R&D Software Engineer at NetApp, where he focuses on the development of model editors for system management and infrastructure automation. His current research revolves around model-based software engineering, domain-specific languages, model editors and infrastructure as code. You can contact the author at ionut.predoaia@york.ac.uk.

Dimitris Kolovos is a Professor of Software Engineering in the Department of Computer Science at the University of York, where he researches and teaches automated and model-driven software engineering. He is also an Eclipse Foundation committer, leading the development of the open-source Epsilon model-driven software engineering platform, and an editor of the Software and Systems Modelling journal. He has co-authored more than 150 peer-reviewed papers and his research has been supported by the European Commission, UK's Engineering and Physical Sciences Research Council (EPSRC), InnovateUK and by companies such as Rolls-Royce and IBM. You can contact the author at dimitris.kolovos@york.ac.uk.

Matthias Lenk During his time at NetApp, Matthias Lenk held the role of a Global Technology Strategist and focused on cloud and data services for the automotive industry. His academic background is within model-driven software development and model transformations in the domain of VR/AR. You can contact the author at matthias.lenk@posteo.net.

Antonio García-Domínguez is a Lecturer at the Department of Computer Science of the University of York. Antonio's main research interest is model-driven software engineering, with lines of work on scalable model management and runtime models for explainability of self-adaptive systems. In addition to over 60 publications across international conferences, journals, and book chapters, Antonio is a core contributor to the Eclipse Epsilon model management languages and tools, and leads the Eclipse Hawk model indexing project. You can contact the author at a.garcia-dominguez@york.ac.uk.