Textual Modelling Embedded into Graphical Modelling

Markus Scheidgen

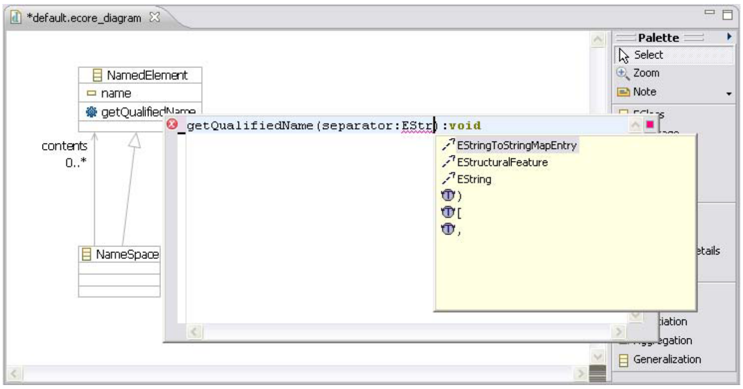
Department of Computer Science, Humboldt Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
`scheidge@informatik.hu-berlin.de`

Abstract. Although labelled graphical, many modelling languages represent important model parts as structured text. We benefit from sophisticated text editors when we use programming languages, but we neglect the same technology when we edit the textual parts of graphical models. Recent advances in generative engineering of textual model editors make the development of such sophisticated text editors practical, even for the smallest textual constructs of graphical languages. In this paper, we present techniques to embed textual model editors into graphical model editors and prove our approach for EMF-based textual editors and graphical editors created with GMF.

1 Introduction

In the past, the superiority of (purely) graphical representations was widely assumed at first and often challenged [1,2] later. Moher et al., for example, concluded in [3]: *"Not only is no single representation best for all kinds of programs, no single representation is [...] even best for all tasks involving the same program."* Today, graphical modelling languages and domain-specific languages (DSLs) often use a mixed form of representation: they use diagrams to represent structures visually, while other elements are represented textually. Examples for textual elements are signatures in UML class diagrams, mathematical expressions in many DSLs, OCL expressions used in other modelling languages, and many programming constructs of SDL [4].

Existing graphical editors address textual model parts poorly. The OCL editors of many UML tools, for example, barely provide syntactical checks and keyword highlighting. As a result, modellers produce errors in OCL constraints: errors that stay unnoticed until later processing, errors that when finally noticed are hard to relate to the OCL constraint parts that caused them. For other constructs, like operation signatures in UML class diagrams, editors often provide no textual editing capabilities at all. So editor users have to use big amounts of clicks to create signatures with graphical editing means. This process is slower and less intuitive than writing the signature down. As a general conclusion, editing the textual parts of models is less efficient than existing editor technology for programming languages would allow.



```
_getQualifiedName(separator:EStr) :void
```

```
EStringToStringMapEntry
EStructuralFeature
EString
)
[
,
```

Fig. 1. The Ecore GMF editor with an embedded textual model editor

This programming technology includes modern programming environments with highly capable language-dependent text editors. These editors allow far more efficient programming than the plain text editors that were used before and that are still used for editing textual parts in graphical models. These modern program editors are complex and have to be built for each language independently. This renders the manual development of such editors too expensive for the textual parts of graphical languages, especially those of DSLs. However, recent research [5,6,7,8,9] utilizes generative engineering techniques to make text editor development for small applications practical. *Language engineers* only provide a high-level textual notation descriptions written in a corresponding meta-language, and meta-tools can automatically generate editors from those descriptions. The usual feature set of such generated editors comprises syntax highlighting, outline views, annotation of syntactical and semantic errors, occurrence markings, content-assist, and code formatting.

In this paper, we present techniques to combine textual modelling with graphical modelling, to embed textual model editors into graphical editors[1]. Editor users open an *embedded text editor* by clicking on an element within the graphical *host editor*. The *embedded editor* is shown in a small overlay window, positioned at the selected element (see Fig. 1). Embedded editors have all the editing capabilities known from modern programming environments. To implement our approach, we use our textual editing framework TEF [11] and create embedded textual editors for graphical editors developed with the Eclipse Graphical Modelling Framework (GMF). TEF allows to create textual notations for arbitrary EMF meta-models. This includes meta-models with existing GMF editors.

As potential impact, our work can enhance the effectiveness of graphical modelling with languages that partially rely textual representations. Furthermore, it encourages the use of domain-specific modelling, because it allows the efficient

[1] The applicability of the presented techniques is not limited to graphical editors, but to editors based on the *Model View Controller (MVC)* pattern [10]. This also includes tree-based model editors, which are very popular in EMF-based projects.

development of DSL tools that combine the graphical and textual modelling paradigm. In general, this work could provide the tooling for a new breed of languages that combine the visualisation and editing advantages of both graphical notations and programming languages.

The remainder of this paper is structured as follows. The next section gives an introduction to textual modelling. After that, section 3 discusses the problems associated with embedding editors in general. In section 4, we realise our approach and implement it for several example languages. The paper is closed with related work and conclusions in sections 5 and 6.

2 Textual Modelling and Generative Engineering of Textual Model Editors

Textual modelling represents models with textual representations; modellers edit models by creating or changing a string of characters. We call the usage of languages with textual notations *textual modelling* and the process of using corresponding editors *textual model editing*. The necessary textual editing tools can be generatively engineered using *textual model editing frameworks*. These frameworks combine meta-languages for textual notations and corresponding meta-tools. Language engineers describe notations and meta-tools generate feature rich textual model editors automatically.

Before we explain how textual modelling can be embedded into graphical modelling, we use this section to introduce general concepts of textual modelling and the generative engineering of textual model editors. We accumulated this information from many existing but similar approaches [5,7,9,12]. Some technical details and the examples are specific to our own textual modelling framework TEF, which we also use to technically realise embedded textual modelling later in this paper.

2.1 Models and Their Representations, Corresponding Meta-models and Notations

We use the term *model* to refer to an abstract structure, where possible elements of this structure are predetermined by a *meta-model*. Thus, a model is always an instance of a meta-model. To read and modify a model, it needs to be represented, for example, graphically or textually. Possible *representations* are

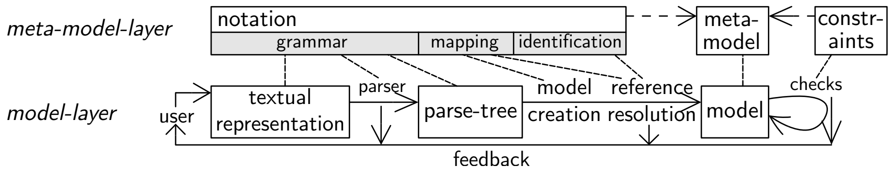


Fig. 2. The background parsing strategy and involved artefacts

defined by a *notation*. In the same way a model is an instance of a meta-model, a model representation is an instance of a notation.

A textual notation (used for the considered textual model editing technology) consists of a context-free grammar, and a mapping that relates this grammar to a meta-model. The grammar defines a set of syntactical correct representations. The mapping identifies the model corresponding to a representation. Fig. 3 shows an example notation. This notation for the Ecore language is mapped to the Ecore meta-model. A possible instance of this notation is shown in the top left of Fig. 4, the corresponding Ecore model is shown in its graphical representation on the top right of the same figure.

The notation in Fig. 3 is written for our TEF framework. It is a combination of a context-free grammar with EBNF elements, augmented with mappings to a meta-model. All string literals function as fixed terminals; all capitalized symbols are morphem classes for integers or identifiers; everything else are non-terminals. The bold printed elements relate grammar elements to meta-model classes and features. If such a meta-model relation is attached to the left-hand-side of a rule, the rule is related to a meta-model class; if attached to a right-hand-side symbol, the right-hand-side rule part is related to a feature. We distinguish between different kinds of meta-model relations: *element* relations relate to classes, *composite* relations relate to attributes or other compositional structural features, and *reference* relations relate to non-compositional structural features.[2]

2.2 The Background Parsing Strategy

Background parsing is a strategy to technically realise textual editing for context-free grammar based notations, used by existing textual model editors (see Fig. 2). Background parsing is a circular process, containing four steps. First, the user edits text as in a regular text editor. Second, the inserted text is parsed according to the notation's grammar. Third, a model is created from the resulting parse-tree based on a given grammar to meta-model mapping. Finally, language constraints are used to check the model. Errors in representation or resulting model can arise in all steps and are reported back to the user. Otherwise, the user is unaware of the parsing process, and continuous repetition gives the impression that the user edits the model directly. As opposed to other editing strategies, background parsing does not change the model, but creates a completely new model that replaces the current model, in each repetition.

[2] The terms *compositional* and *non-compositional*, hereby, refer to structural features that imply and not imply exclusive ownership. The corresponding structural features can carry other model elements as values. One model element used as a value in a compositional feature of another element becomes part of the other element. The part is called a *component* of the owning element; the owning element is called *container* respectively. Each model element can be the component of one container only, i.e. it can be a value in the structural features of one single model element at best. The component-container relationship is called *composition*. Cycles in composition are not allowed; composition forms trees within models. The roots of these trees are model elements without container; the leafs are elements without components. [13]

2.3 Creating Models from Parse-Trees

The result of parsing a textual representation is a syntax-tree or *parse-tree*. Each node in a parse-tree is an instance of the grammar rule that was used to produce the node. To create a model from a parse-tree, editors perform two depth-first traversals on the tree.

In the first run, editors use the element relations attached to a node's rule to create an instance of the corresponding meta-model class. This instance becomes the value represented by this node. It also serves as the context for traversing the children of this node. Morphems create corresponding primitive values, e.g. integers or strings. During the same traversal, we use the composite relations to add the values created by the respective child nodes to the referenced features in the actual context object. With this technique, the parse-tree implies composition between model elements. Furthermore, the compositional design of the meta-model must match the alignment of corresponding constructs in the textual notation.

```
syntax(Package) "models/Ecore.ecore" {
  Package:element(EPackage) ->
    "package" IDENTIFIER:composite(name)
    "{" (PackageContents)* "}";

  PackageContents -> Package:composite(eSubpackages);
  PackageContents -> Class:composite(eClassifiers);
  PackageContents -> DataType:composite(eClassifiers);

  Class:element(EClass) ->
    "class" IDENTIFIER:composite(name) (SuperClasses)?
    "{" (ClassContents)* "}";

  SuperClasses -> "extends" ClassRef:reference(eSuperTypes)
    ("," ClassRef:reference(eSuperTypes))*;
  ClassRef:element(EClass) -> IDENTIFIER:composite(name);

  ClassContents -> "attribute" Attribute:composite(eStructuralFeatures);
  ClassContents -> "reference" Reference:composite(eStructuralFeatures);
  ClassContents -> Operation:composite(eOperations);

  Attribute:element(EAttribute) -> IDENTIFIER:composite(name) ":"
        TypeRef:reference(eType) Multiplicity;

  TypeRef:element(EClassifier) -> IDENTIFIER:composite(name);

  Multiplicity -> ("[" INTEGER:composite(lowerBound) ":"
        UNLIMITED_INTEGER:composite(upperBound) "]")?;

  Operation:element(EOperation) -> ...
}
```

Fig. 3. Excerpt from a notation description for the Ecore language

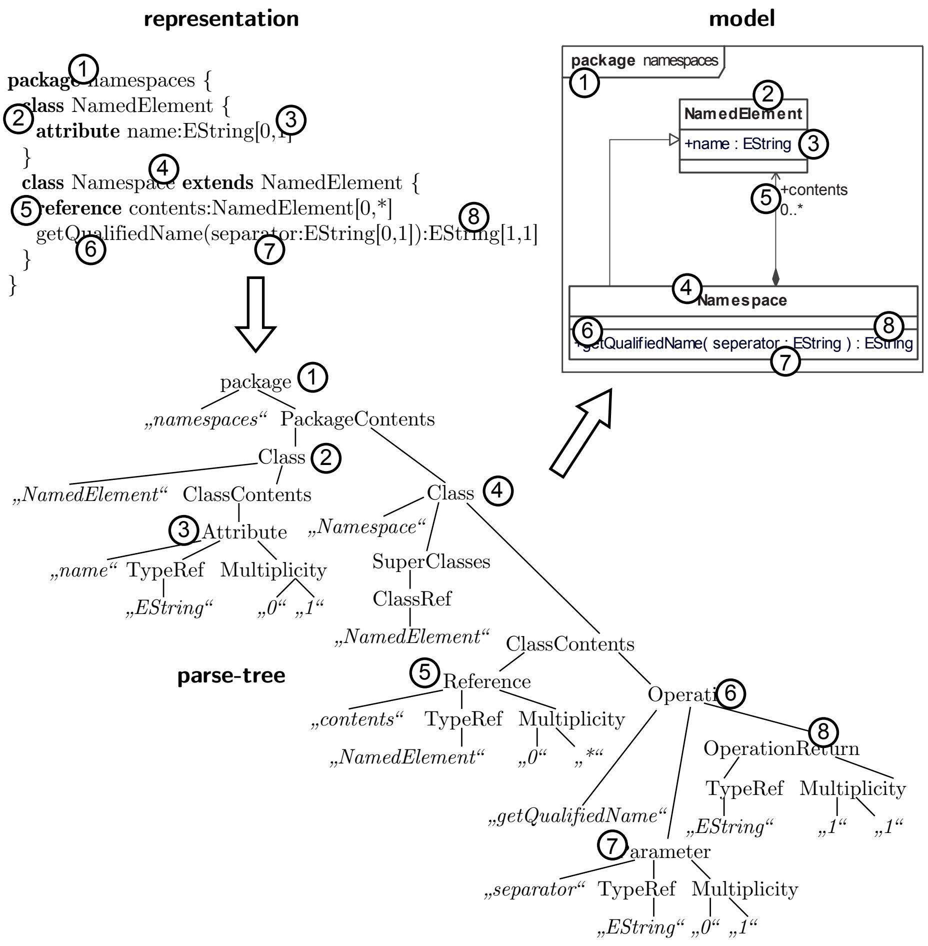


Fig. 4. A representation, parse-tree, and model based on the example Ecore notation

2.4 Identity and Reference Resolution

Before we can understand how editors add feature values for references in the second traversal, we have to understand identification of model elements. The *identity* of a model element is a value that uniquely identifies this element within a model. Each language defines a function that assigns each element of a language instance an identity. Example identities can be the model element itself, the name of the model element, or more complex constructs like full qualified names. An *identifier* is a value used to identify a model element based on the element's identity. In simple languages, identifiers can often be used directly to identify model elements: if an identifier and the identity of a model element are the same, this identifier identifies this model element. In many languages, however, identification is more complex, including name spaces, name hiding, imports, etc.

In those cases, an identifier depends on the context it is used in. The language must define a function that assigns a set of possible global identifiers to an identifier and its context. These global identifiers are then used to find a model element with an identity that matches one of those global identifiers.

Identity and identifiers are language-specific and might vary for model elements of different meta-classes. Textual model editing frameworks can only provide a simple default identification mechanism, i.e. based on a simple identity derived from a model element's meta-class and possible *name* attribute. TEF and other frameworks allow to customize this simple behaviour. Because no specific description mechanisms for identification could be found for existing textual editing frameworks yet, this part of a notation definition has usually to be programmed within the used textual editing framework.

In the second traversal (also called *reference resolution*), the editor goes through parse-tree and model simultaneously. Now, it uses all reference relations to add corresponding values to all non-compositional structural features. This time, it does not use the child nodes' values directly, but uses the child nodes' values as identifiers to resolve the corresponding referenced elements. Because all model elements were created in the first traversal, the referenced model elements must already exist in the model.

3 Embedded Textual Model Editing

Graphical model editors are based on the *Model View Controller* MVC pattern [10]. An MVC editor displays representations for model elements (model) through view objects (view). It offers actions, which allow the user to change model elements directly. Actions are realised in controller objects (controller). Examples for such actions are creating a new model element, modify the value set of a model element's feature, deleting a model element. The representing view objects react to these model changes and always show a representation of the current model. In MVC editors the user does not change the representation, only the model; the representation is just updated to the changed model. From now on, we assume that the host editor is an MVC editor.

We propose the embedded editing process illustrated in Fig. 5: The user selects a model element in the host editor and requests the embedded editor. The embedded editor is opened for the selected model element (1). We call this model element the *edited model element*. The edited model element includes the selected model element itself and all its components. The opened textual model editor creates an initial textual representation for the edited model element (2). The user can now change this representation, and background parsing creates new partial models, i.e. creates new edited model elements (3). The model in the host editor is not changed, until the user commits changes and closes the embedded textual model editor. At this point, the embedded editor replaces the original edited model element in the host editor's model, with the new edited model element created in the last background parsing iteration (4).

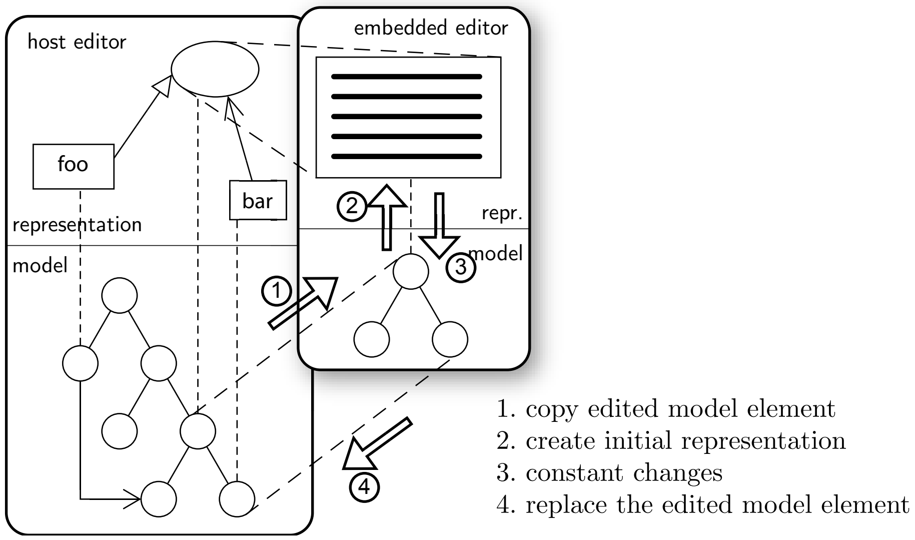

host editor
embedded editor
foo
bar
representation
model
repr.
model
1
2
3
4
1. copy edited model element
2. create initial representation
3. constant changes
4. replace the edited model element

Fig. 5. Steps involved in the embedded textual editing process

There are three problems. First, we need textual model editors for partial models. Obviously, it is necessary to describe partial notations for corresponding partial representations. Second, when the editor is opened, it needs to create an initial textual representation for the selected model part. Finally, when the editor is closed, a new partial model has been created during background parsing. The newly created partial model needs to replace the original edited element. All references and other information associated with the original edited model element have to be preserved.

3.1 Creating Partial Notation Descriptions

Textual model editors rely on textual notations. Whether these notations cover a language's complete meta-model or just parts of it, is irrelevant, as long as the edited models only instantiate those meta-model parts that are covered by the textual notation.

Two solutions are possible: first, language engineers only provide a partial notation for the meta-model elements that they intend to provide embedded textual modelling for. In this case, the engineers have to be sure that they cover all related meta-model elements. Textual editing frameworks can automatically validate this. Second, language engineers provide a complete notation, and the editing framework automatically extracts partial notations for each meta-model element that embedded editing is activated for.

3.2 Initial Textual Representations

To create the initial textual representation, an editor has to reverse the model creating and parsing process: it creates a parse-tree from the edited model element and pretty prints this tree.

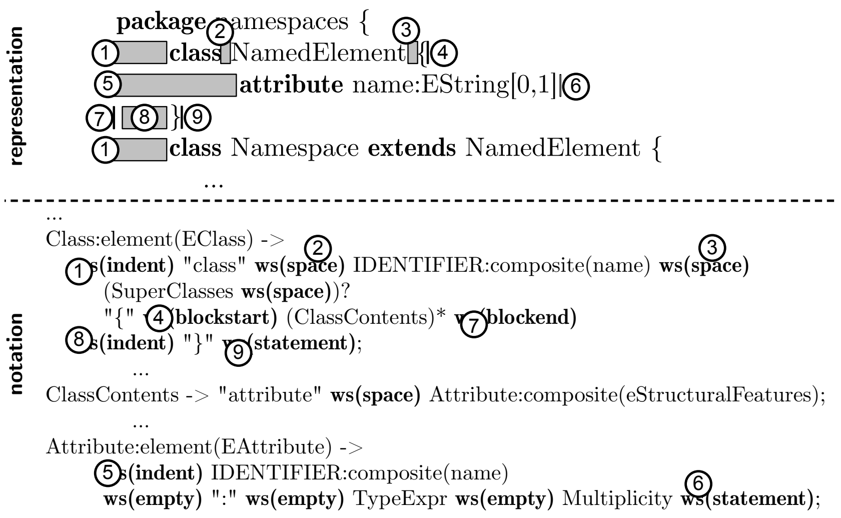
Creating a Parse-tree. To create a parse tree, the editor traverses the model along its composition. For each model element, the editor can determine a set of suitable grammar rules based on the element's meta-class, the values of its features, and the grammar to meta-model mapping. By using a back-tracking strategy, the editor can determine one or more possible parse-trees for a model. Notations that provide different possibilities to represent a single language construct in different ways, will lead to multiple possible parse-trees for the same model. Frameworks can give language engineers the possibility to prioritise grammar rules accordingly. If the editor cannot determine a parse-tree for a model, meta-model, grammar, and grammar to meta-model mapping are inconsistent. For example, a model element name might be optional as defined by the meta-model, but required by the notation: at some point in the model traversal, all possible grammar rules for the element would require the name of the element, but the name cannot be obtained.

Pretty Printing the Parse-tree. Pretty printing a parse-tree is basically straight forward. The only problem are the white-spaces between two tokens. A human readable textual representation needs reasonable white-spaces, i.e. *layout information*, between these tokens. This is a problem with two possible solutions. First, white-spaces originally created by editor users, can be stored within the model. Second, editors can create white-spaces automatically. Disadvantages for storing layout information are that the layout information has to be provided by editor users, and model and meta-model have to be extended with layout information elements. The advantage is that user layouts and all information that users express within layouts (also known as secondary notation [2]) are preserved. The second solution has complementary advantages and disadvantages.

For embedded editing, we propose the automatic generation of white-spaces. The edited text usually only comprises text pieces; white-spaces with hidden information, like empty lines, are not that important. Furthermore, the embedded text editors rely on the modelling facilities of the host editor; storing information beyond the model requires to change the host editor's implementation. And finally, with automatic layout, it is also possible to textually represent models that were not created via a textual representation. Models created with other means than a textual model editor (e.g. the host editor) can also be edited within such an editor.

Automatic layout of textual representations requires white-space clues as part of the textual notation definition. We propose the use of *white-space roles*. Language engineers add white-space roles as symbols to grammar rules. A white-space role determines the *role* that the separation between two tokens plays. Whites-spaces roles are dynamically instantiated with actual white-spaces, when text is created from a model. A component called *layout manager* defines possible white-space roles and is used to instantiate white-space roles. It creates white-spaces for white-space roles in the order they appear within the created textual model representation. The layout manager can instantiate the same white-space role differently, depending on the context the white-space role is used in.

An example: A layout manager for block-layouts as used in programming languages supports the roles *space*, *emtpy*, *statement*, *blockstart*, *blockend*, *indent*.



```
      package namespaces {
      class NamedElement {
        attribute name:EString[0,1]
      }
      class Namespace extends NamedElement {
        ...
```

```
...
Class:element(EClass) ->
    ws(indent) "class" ws(space) IDENTIFIER:composite(name) ws(space)
        (SuperClasses ws(space))?
        "{" ws(blockstart) (ClassContents)* ws(blockend)
    ws(indent) "}" ws(statement);
        ...
ClassContents -> "attribute" ws(space) Attribute:composite(eStructuralFeatures);
        ...
Attribute:element(EAttribute) ->
        ws(indent) IDENTIFIER:composite(name)
        ws(empty) ":" ws(empty) TypeExpr ws(empty) Multiplicity ws(statement);
```

Fig. 6. An example text with white-spaces for an example notation with white-space roles

This manager instantiates each *space* with a space and each *empty* with an empty string. But, if the manager detects that a single line of code becomes too long, the layout manager can also instatiate both roles with a return and a followed proper identation. The manager uses the *blockstart* and *blockend* roles to determine how to instantiate an *indent*. It increases the indentation size when a *blockstart* is instantiated, and decreases it, when a *blockend* is instatiated. Fig. 6 shows an example representation and corresponding notation with white-space roles for the Ecore language based on the block-layout manager.

3.3 Committing Model Changes

Problems caused by different editing paradigms. The host editor changes the model with small actions that only affect single or very few model elements. Opposite to the MVC host editor, the embedded textual model editor is based on background parsing and creates complete new model elements for each representation change. This causes two problems. First, other model elements that are not part of the edited model element might reference the edited model element or parts of it. These references break when the original edited model element is replaced by a new one. Second, the edited model element might contain information that is not represented; this information will be lost, because it is not part of the model element created through background parsing.

In today's modelling frameworks, e.g. EMF, we know all the references into the edited element and its parts, and we can simply reassign these references to the replacement model element and its parts. This would solve the first problem. We could also merge changes manifested in the newly created model element into the original model element. This would only update the original edited model element and not replace it. This would solve both problems. Anyway, both solutions require identification: the editor has to access whether a model element in the original model element is meant to be the same as a corresponding model element part of the edited (newly created) model element. The editor can achieve this based on the elements' identity. This is obviously language-specific, and identification has to be defined for each language (see section 2).

With identification, the editor can tell whether two model elements have the same identity, and realising the first problem solution becomes very easy. The editor takes all references into the original edited model element, determines the identity of the referenced model elements within the original editor model element, searches for a model element with the same identity within the newly created model element, and reassigns the reference. The second problem solution requires some sort of algorithm that navigates both, the edited model element and the newly created model element, simultaneously along the model elements' composition. The merge algorithm has to compare the model elements feature by feature based on their identity, and transcribes all differences into the original edited model element. Deeper discussions about model merging is outside of this paper's scope; model merging algorithms and techniques are described in [14,15].

One problem remains: this problem occurs, if the user changes the text representation in a way that the identity of an element changes. Using the background parsing strategy, it is not clear what the user's intentions are. Did the user want to change, e.g., the name of a model element, or did he want to actually replace a model element, with a new element. The editor can only assume that the user wanted to create a new element. One way to solve this problems is to give the user the possibility to express his intention, e.g. provide a refactoring mechanism that allows to rename a model element.

Problems Caused by Different Undo/redo Paradigms. A convenient feature of model editors is the possibility to undo and redo model changes. This needs to be preserved for model changes in embedded text editors. In MVC editors, model changes are encapsulated in command objects, which allow to execute single model changes, and to reverse the execution of single model changes. Commands for executed model changes are stored in a command stack, which the editor uses for undo/redo. This is different in a textual model editor based on background parsing, where users change a string of characters. User actions are represented as replacements on that string. Undo/redo is based on a stack of string replacements. Embedded textual model editors and their graphical host editors obviously use incompatible representations for model changes.

We propose the following solution. Embedded text editors offer string replacement based undo/redo during textual editing. When the user closes the embedded editor and commits the textual changes, the necessary actions to replace the original edited model element are encapsulated into a single command, which is then stacked into the host editors undo/redo facility. This is a compromise: it allows to undo whole textual editing scenes, but does not allow to undo all the intermediate textual editing steps once the embedded editor is closed.

4 Realisation and Experiences

4.1 A Framework for Embedded Textual Model Editors

We created an EMF-based ([13]) textual editing framework for the Eclipse platform called Textual Editing Framework (TEF) [11]. This framework allows to describe textual model editors that use the background parsing strategy. Editors can be automatically derived from notation descriptions and support usual modern text editor features: syntax highlighting, error annotations, content assist [16], outline view, occurrences, and smart navigation. An example notation description is shown in Fig. 3.

We extended TEF for the development of embedded editors. Embedded editors can be created for EMF generated tree-based editors and editors created with the Graphical Modelling Framework (GMF) [17]. These embedded editors do not require to change the host editor. In theory, TEF should work for all EMF-based MVC host editors. To use TEF for embedded editors, language engineers provide a notation description for those meta-model elements that embedded textual editing is desired for. TEF automatically generates the embedded editors and provides so called object-contributions for corresponding EMF objects. These object-contributions manifest as context menu items in the host editor. With these menu items, users can open an embedded text editor for the selected model element. The embedded editor is a full fledged TEF editor providing all its features, except for the outline view, which is still showing the host editors outline. The embedded editor will only show the textual representation of the edited model element. The embedded text editor can be closed in two ways. One way indicates cancellation (by clicking somewhere into the host editor); the other way commits the changes made (pressing shift-enter).

We used the following problem solutions in TEF, which automatically creates partial notation descriptions by reducing a given notation for the specific edited model element dynamically, when the editor is opened. TEF editors create initial representations based on pretty printing with automatic white-space generation using layout managers. Embedded editors commit model changes by creating one single compound command that is added to the host editors command stack to preserve the host editors undo/redo capability. This command contains sub-commands that replace the original edited element and reassign all broken references based on either TEF's default identification or a language-specific identification mechanism. We plan to implement merging of newly created edited model element and original model element as future work.

4.2 Textual Editing of Ecore Models

We used TEF to develop a textual notation for Ecore and generated embedded textual editing for the graphical Ecore GMF editor and the standard tree-based Ecore editor. We wanted a more convenient editing of signatures for attributes, references, and operations. With the textual editing capabilities this becomes indeed more convenient and, e.g., renders the process of creating an operation with many parameters more efficient.

The work on the Ecore editors affirmed a few of the mentioned problems. First, many elements in the Ecore language carry information drawn from many side aspects of meta-modelling, such as parameters for code generation or XMI generation. Including all this information in the textual notation, would render it very cumbersome. Omitting this information in the textual notation, however, causes the loss of this information, when the corresponding model parts are edited textually. We hope to eliminate this problem by applying a model merging approach to update the original edited model element instead of replacing it. Second, a textual notation for Ecore needs complex identification constructs to realise textual references. Identification in Ecore has to rely on namespaces, imports, local and full qualified names. None of those constructs were defined in the language itself, and had to be invented on top of the actual Ecore language. We had to augment the automatic generated editing facilities, with manual implementations that describe such identification constructs.

4.3 An OCL Editor Integrated into Other Model Editors

The OCL constraint language is often used in conjunction with languages for object-oriented structures, or the behaviour of such structures. Hence, OCL expressions are often attached to the graphical notations of languages like UML, MOF, or Ecore. Therefore, editors for those languages should support OCL editing, but they usually only do by means of basic string based text editing.

We used TEF to develop an OCL editor based on the MDT OCL project. As an example, we integrated this editor with the tree-based Ecore editor: The EMF validation framework requires OCL constraints stored in Ecore annotations, because Ecore itself does not support the storage of OCL constraints. Because this only allows to store OCL constraints in their textual representation as strings, the embedded textual OCL editor is actually a normal text editor, which only uses background parsing to create internal OCL models to support advanced editor features, i.e. code-completion and error annotations. This makes committing the changes to OCL constraints particularly easy, because the embedded editor only has to replace a string annotation in the host-editors Ecore model. Another example application that we integrated the OCL editor into is the graphical editor for the UML activity-based action language in [18,19].

4.4 Editing for Mathematical Expressions in DSLs

Many of today's DSLs are developed based on EMF and instances of these languages are consequently edited using EMFs default generated tree-based model editors. This is fine for most parts of these languages, but can become tiresome, if models contain mathematical expressions. Because mathematics is commonly used to express computation, such expressions are part of many languages.

We used TEF to develop a simple straight forward notation for a simple straight forward expression meta-model. This expression meta-model and notations is a blueprint for integrating sophisticated editing capabilities for expressions into DSLs. We used this to realise expressions in a domain-specific language for the definition of cellular automatons, used to predict the spread of natural disasters like floods or fire.

5 Related Work

Work on textual notations based on meta-modelling with MOF includes Alanen et al. [20], Scheidgen et al. [21], Wimmer et al. [12]. This basic research was later utilised in frameworks for textual model editors. These are either based on existing meta-models (TCS [7], TCSSL [9], MontiCore [5]), or they generate meta-models or other parse-tree representations generated from the notations (xText [22], Safari [6]). Those frameworks however, only support the editing of text files. Models have to be created from those text files separately. Furthermore, pretty printing capabilities, if supported, are not directly integrated into the editors (because they are only text editors). This makes it hard to facilitate those frameworks for embedded textual editing, because these editors can not create an initial representation.

The GMF framework itself, provides some very simple means to describe structured text. It allows to create simple templates that assign different portions of a text to different object features. These simple templates allow less than regular complexity, and are therefore inadequate for many textual language constructs. First premature steps to describe the relations between graphical notations and textual notations have been made by Tveit et al. [23].

Background parsing, its notation meta-languages, and used algorithms are inspired by attributed grammars as described in [24]. Besides using context-free grammar and background parsing, textual modelling can also be conducted using the MVC pattern. MVC is used to realise Intentional Programming [25] and the Meta Programming System (MPS) [26]. Because the editing paradigm is the same than in graphical editors, it is thinkable that these frameworks can be integrated into graphical editors as well, maybe with more natural solutions to most of the presented problems. However, using MVC, only allows to create models based on simple actions. This way, you have to create, e.g., an expression as if you would create its parse-tree: from top to bottom. This is the same kind of limitation that is already imposed upon graphical editors, and this is exactly why we want to use textual modelling in the first place.

6 Conclusions and Future Work

The work presented in this paper showed that textual model editors can be embedded into graphical editors. It also showed that this can be done efficiently with generative development based on already existing technology. Furthermore, existing graphical editors do not necessarily have to be changed to embed textual editors; embedded editors can be provided as an add-on feature. Although, only formal studies can proo that embedded textual modelling raises productivity, we can assume increased productivity from the effect that modern text editors already had on programming.

There are a few issues that we suggest for further work. Problematic is that background parsing makes it hard to read the user's intention. A further problem are changes that alter a model element's identity and might result in unintended effects. In today's programming world, this is solved with refactoring. Textual editing frameworks need to allow the generative engineering of such facilities for textual editors (or model editors in general).

Furthermore, we need to explore the different grades of notation and editor integration. In this paper, we suggested to provide notation descriptions for both the graphical and the textual notation separately, which allows to use both editors (host and embedded) on arbitrary model elements. But, in many cases it would be more natural to integrate both notation description into a single notation description. This requires better integrated description languages and editing frameworks, but would allow for more concise, coherent notation descriptions and a more seamless overall modelling experience. On the technical site, we should explore other possibilities to integrate editors into each other. For example, it would be desirable to have the textual editor widget directly contained in the corresponding graphical widget. This makes additional overlay windows superfluous and would create a more fluid editing experience. Furthermore, this allows to see syntactical highlighting and error annotations directly in the graphical model view.

References

1. Green, T.R.G., Petre, M.: When Visual Programs are Harder to Read than Textual Programs. In: Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics) (1992)
2. Petre, M.: Why looking isn't always seeing: readership skills and graphical programming. Commun. ACM 38(6) (1995)
3. Moher, T.G., Mak, D.C., Blumenthal, B., Leventhal, L.M.: Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets. In: Empirical Studies of Programmers - Fifth Workshop (1993)
4. ITU-T: ITU-T Recommendation Z.100: Specification and Description Language (SDL). International Telecommunication Union (2002)
5. Krahn, H., Rumpe, B., Völkel, S.: Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735. Springer, Heidelberg (2007)
6. Charles, P., Dolby, J., Fuhrer, R.M., Sutton, S.M., Sutton, J.S.M., Vaziri, M.: SAFARI: a meta-tooling framework for generating language-specific IDE's. In: OOPSLA 2006: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (2006)
7. Jouault, F., Bézivin, J., Kurtev, I.: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: GPCE 2006: Proceedings of the 5th International Conference on Generative Programming and Component Engineering (2006)
8. Kleppe, A.: Towards the Generation of a Text-Based IDE from a Language Metamodel. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530. Springer, Heidelberg (2007)

9. Muller, P.A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J.M.: Model-Driven Analysis and Synthesis of Concrete Syntax. In: 9th Intern. Conf. on Model Driven Engineering Languages and Systems (2006)
10. Holub, A.: Building user interfaces for object-oriented systems. JavaWorld (1999)
11. Homepage: Textual Editing Framework, `http://tef.berlios.de`
12. Wimmer, M., Kramler, G.: Bridging Grammarware and Modelware. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713. Springer, Heidelberg (2005)
13. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework (The Eclipse Series). Addison-Wesley Professional, Reading (2003)
14. Zito, A., Diskin, Z., Dingel, J.: Package Merge in UML 2: Practice vs. Theory? In: 9th Intern. Conf. on Model Driven Engineering Languages and Systems (2006)
15. Kolovos, D.S., Paige, R.F., Polack, F.: Merging Models with the Epsilon Merging Language (EML). In: 9th Intern. Conf. on Model Driven Engineering Languages and Systems (2006)
16. Scheidgen, M.: Integrating Content-Assist into Textual Model Editors. In: Modellierung 2008. LNI (2008)
17. Homepage: Graphical Modelling Framework, `http://www.eclipse.org/gmf/`
18. Scheidgen, M., Fischer, J.: Human Comprehensible and Machine Processable Specifications of Operational Semantics. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530. Springer, Heidelberg (2007)
19. Eichler, H., Soden, M.: An Approach to use Executable Models for Testing. In: Enterprise Modelling and Information Systems Architecture. LNI (2007)
20. Alanen, M., Porres, I.: A Relation between Context-Free Grammars and Meta Object Facility Metamodels. Technical report, TUCS (2004)
21. Fischer, J., Piefel, M., Scheidgen, M.: A Metamodel for SDL-2000 in the Context of Metamodelling ULF. In: System Analysis and Modeling, 4th International SDL and MSC Workshop, SAM (2004)
22. Homepage: openArchitectureWare, `http://www.openarchitectureware.org`
23. Prinz, A., Scheidgen, M., Tveit, M.S.: A Model-Based Standard for SDL. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. 4745. Springer, Heidelberg (2007)
24. Knuth, D.E.: Semantics of Context-Free Languages. Theory of Computing Systems 2 (1968)
25. Simonyi, C.: The death of computer languages, the birth of Intentional Programming. Technical report, Microsoft Research (1995)
26. Dmitriev, S.: Language Oriented Programming: The Next Programming Paradigm. onBoard (November 2004)