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. Today's graphical modelling languages, despite using symbols and connections, represent large model parts as structured text. We benefit from sophistic 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 allow to create such sophisticated text editors more efficiently. Therefore, textual modelling becomes practical for the textual parts of graphical models. In this paper, we present the necessary techniques to embed generated EMF-based textual model editors into graphical editors created with GMF and tree-based editors generated with EMF.

1 Introduction

Today's graphical modelling and domain specific languages represent large model parts as structured text. Even though, those languages use symbols and connection to represent model structures visually, there are typical model elements that should only be represented as text. Examples for these elements are signatures in UML class diagrams or meta-models, mathematical expressions in many DSLs, the many constructs in SDL (a language that is a mix of graphical modelling and programming language), or the combination of structure modelling and OCL.

The last example manifests in UML tools that allow to add OCL constraints to UML models, but only offer simple text editors. These OCL editors barely provide syntactical checks and keyword highlighting. As a result, model editor users produce many 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. In other examples, like operation signatures in UML class diagrams, we often have no textual editing capabilities at all. So we *click* signatures together. 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 editor technology for programming languages would allow.

To program, we use modern programming environments with highly capable language dependent text editors. These last generation editors allow us to program far more efficiently than we programmed with plain text editors that

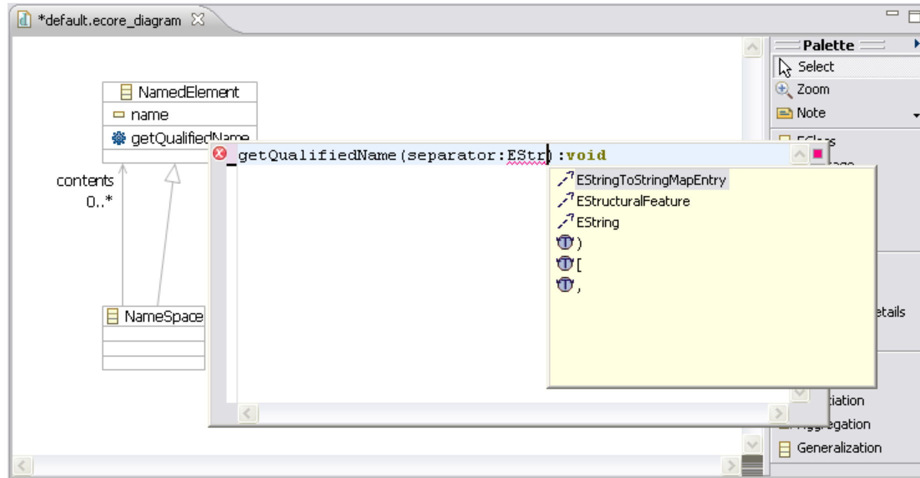

Fig. 1. The ecore GMF editor with an embedded textual model editor.

were used before and that are still used for editing text parts in graphical model editors.

These modern text editors are complex pieces of software and have to be build for each language with extensive efforts. Recent research, however, provides us with meta-languages and meta-tools for the description and automatic generation of such highly capable text editors. An editor or *language engineer* only provides descriptions for language and textual notation. Meta-tools can then generate editors from those high-level language descriptions automatically. The usual feature set of these editors comprises syntax highlighting, outline views, annotation of syntactical and semantic errors, occurrence markings, content-assist, code formatting, and so on. These editors can be used to edit text or create models as instances of according meta-models. We call the usage of languages with textual notations *textual modelling* and the process of using such editors *textual model editing*. The combination of meta-language for textual notations and according meta-tools is called a *textual editing framework*.

In this paper, we combine textual modelling with graphical modelling; we embed generated textual model editors into graphical editors (or any other editor based on the model view controller pattern for that matter). Editor users can open *embedded text editors* by clicking on an element within the graphical *host* editor. These *embedded editor* are, for example, shown in small overlay windows, positioned at the selected element (see figure 1). Within these embedded editors, the user gets all the features known from modern programming environments. We use our textual editing framework (TEF) to create embedded editors for graphical editors developed with the eclipse graphical modelling framework (GMF). TEF allows to create textual notation for arbitrary EMF meta-models. We can take a meta-model that we already have a GMF editor for, and create an additional textual notation for that meta-model.

This work has the potential to enhance the effectiveness of graphical modelling with languages that rely on textual representations of expressions, signatures, etc. Furthermore, it encourages the use of domain specific modelling, since it allows the efficient development of DSLs that combine the graphical and textual modelling paradigms. 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 2 gives background information about textual modelling, and introduces the background parsing strategy which is used by most textual editing frameworks. The next section 3 introduces three problems that arise, when we integrate textual model editors into other editors. We discuss the possible solutions to these problems. In section 4, we describe the realisation of embedded textual modelling based on our textual editing framework TEF, the eclipse modelling framework EMF, and the eclipse Graphical Modelling Framework GMF, and several example projects. The paper is closed with related work 5 and conclusions 6.

2 State of the Art in Textual Model Editing

Textual model editing describes the process changing a model by changing its textual representation. The necessary textual editing tools are developed by language engineers using textual model editing frameworks. These frameworks provide a notation description language, which is used to define textual notations. The efforts of describing a textual notation is paid off in feature rich textual model editors automatically generated. Depending on the used framework, these editors support syntax highlighting, outline views, error annotations, content-assist, etc., and of course they can create a model from a textual representation.

Before we embed textual modelling into graphical modelling, we use this section to introduce textual model editing concepts based on our framework TEF. More about textual modelling and editing frameworks can be learned from [1,2,3,4].

Models and meta-models. We use the term *model* to refer to an abstract structure, where possible elements of this structure are predetermined through a meta-model. A meta-model thus defines a set of possible models. Three things are important about the term model in this paper. One, a model is always an instance of a language; a model therefore is always an instance of a meta-model. Two, we don't care about the semantics and pragmatics of a model or its language. Therefore, we use the term for both models that are an abstraction of something, like the UML model of a software system, and for models that describe a software system in all detail, like the Java code that constitutes a software system. Three, a model needs *representation.* This can be a visual representation as a diagram or a piece of text. In the same way a model is an instance of a meta-model, a model representation is an instance of a *notation.* Meta-model and notation have to be related.

```
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);

  DataType:element(EDataType) -> ...

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

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

  Reference:element(EReference) -> ...;

  Operation:element(EOperation) -> IDENTIFIER:composite(name)
    "(" (Parameter:composite(eParameters)
          ("," Parameter:composite(eParameters))*)? ")"
    ":" OperationReturn;

  OperationReturn -> TypeRef:reference(eType) Multiplicity;
  OperationReturn -> "void";

  Parameter:element(EParameter) -> IDENTIFIER:composite(name) ":"
        TypeRef:reference(eType) Multiplicity;

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

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

Fig. 2. Example notation descriptions for the Ecore language.

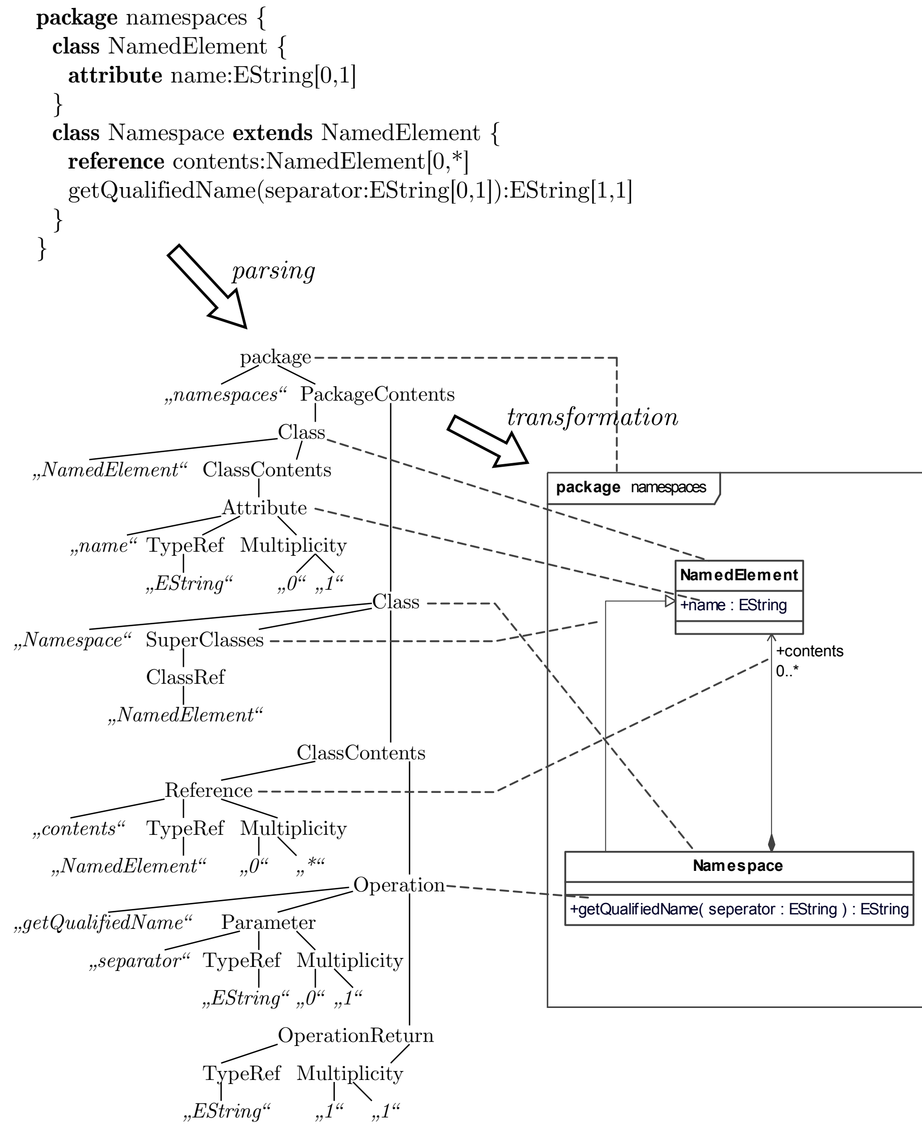

Fig. 3. An example background parsing within an textual ecore editor.

Textual representations and notations. Textual modelling uses text to represent models. Textual model editors show the edited model in its textual representation and allow users to change the edited model by changing its textual representation. Textual modelling is based on textual notations. A textual notation consists of a context-free grammar, and a mapping that relates this grammar with the language's meta-model. The grammar defines a set of syntactical correct textual representations. The mapping identifies the model corresponding

to a representation. Figure 2 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 of figure 3. This textual notation instance is the representation for the ecore model represented graphically on the right side of the same figure.

Over the time, several languages for the definition of textual model notations have been created (refer to the related works section). The notation shown in figure 2 is written for our TEF framework. It is basically a combination of an context-free grammar with BNF elements, augmented with mappings to a meta-model. All string literals function as fixed terminals; all capitalized symbols are morphems for integers or identifiers; everything else are non-terminals. The bold printed elements map grammar elements to meta-model classes and features. If sucha 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-hands-side symbol, this right-hand-side rule part is related to a feature. We distinguish between different kinds of meta-model relations: (1) *element* relations relate to classes, *composite* relations relate to attributes or other compositional structural features, and *reference* references refer to non compositional structural features.

The background parsing process. Most existing textual model editors use *background parsing*. Background parsing is a strategy to technically realise textual editing for context-free grammar based notations. Background parsing is done in three continuously repeated steps. First, the user edits text using a normal 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 meta-model and grammar to meta-model mapping. The user is unaware of the parsing process, and continuous repetition gives the impression that the user edits the model directly. Background parsing, as opposed to other editing strategies, does not change the model. Instead, is creates a completely new model with each change to the textual representation. This distinct characteristic will become important later.

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 of the tree.

In the first run, editors use the element relations attached to a node's rule to create an instance of the according meta-model class. The created object becomes the value represented by this node. It also serves as the context for traversing the children of this node. Morphems create according primitive values, e.g. integers or strings. During the same traversal, we use the composite relations, attached to right-hand-side symbols to add the values created by the respective child nodes to the referenced features in the actual context objects.

In the second traversal (also called *reference resolution*), editors go through the parse-tree and created model simultaneously. Now, it uses all reference relations to add according values to all non compositional structural features. This

time, it does not use the child node value directly, but uses the child node's value as an identifier to resolve the according referenced model element. Since all model elements were created in the first traversal, the referenced model element must exist in the model. Unfortunately, model element identity is different from language to language. Some languages simply use a single model element feature to uniquely identify an element, other languages use complex qualified identifiers, with namespaces, name hiding, imports, etc. Textual model editing frameworks can only provide a simple default reference resolution, i.e. based on a simple identity derived from an model elements meta-class and possible *name* attribute. This simple default behaviour usually has to be customized by language engineers. Technically, this customization of identity and reference resolution is also part of the notation definition. Since, no specific description language or mechanism could be found for existing textual editing framework yet, this part of a notation definition has usually to be programmed within the used textual editing framework.

3 Embedded Textual Model Editing

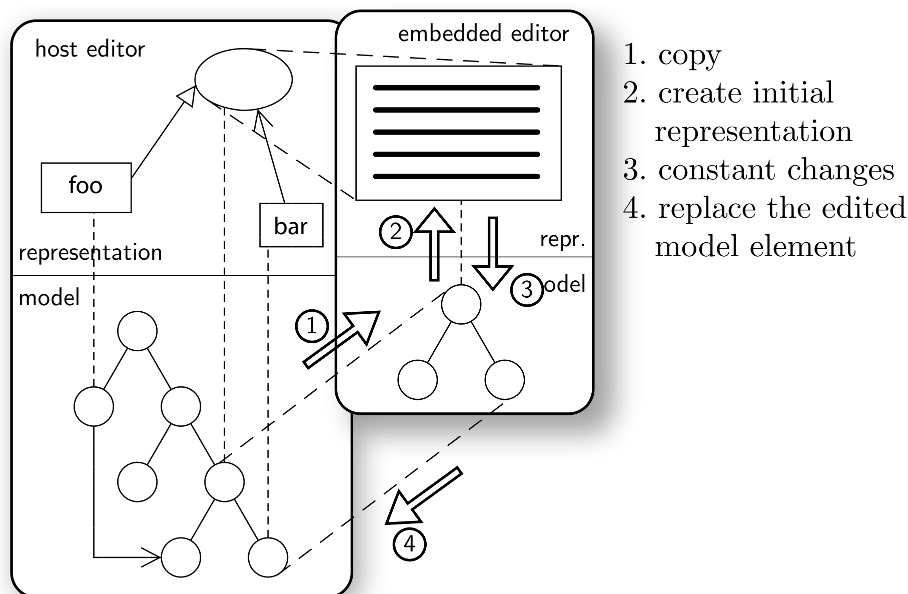

Fig. 4. Steps involved in the embedded textual editing process.

Assumed we have a host editor based on the model view controller pattern. This means that the editor displays representations for model elements through

view objects. It offers actions, which allow the user to change model elements directly. The representing view objects react to these model changes and always show a representation of the current model. Most model editors, especially graphical model editors, use this pattern.

Based on such a host editor, we propose that embedded textual model editors work as follows. 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, over time, 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 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). This embedded editing process is illustrated in figure 4.

There are three problems. One, we need textual model editors for partial models. Obviously, it is necessary to describe the textual notation for according partial notations. Two, when the editor is opened, it needs to create an initial textual representation from the model part edited. Three, when the editor is closed, a new partial model was created during background parsing and is about 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

The problem: 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: Firstly, language engineers 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 to cover all related meta-model elements. This can be automatic validated by a textual model editing framework, which could analyse partial notations in relation to meta-models. Secondly, language engineers provide a complete notation, and the editing framework automatically slices this notation into partial notations for each meta-model element that embedded editing is intended for.

3.2 Formatting the textual representations

The problem: To create the initial textual representation, an editor can reverse the parsing and model creation process: it creates a parse-tree from the edited model element and pretty prints this tree. But, since the notation description

only provides a grammar, destined to parse a stream of tokens, the notation only provides the necessary information to pretty print a stream of tokens. A human readable textual representation, however, also needs whitespaces (or *layout information*) between these tokens.

This is a problem with two possible solutions. Firstly, whitespace, originally created by editor users, can be stored within the model. Secondly, editors can create whitespaces automatically. Both solutions have advantages and disadvantages.

The first solution is very popular with graphical model editors, where layout information is extensive and hard to create automatically. Layout information is always obtained from the user, and it is preserved as part of the model. Disadvantages of this approach are that the layout information has, at one point, to be provided by editor users, and model and meta-model have to be extended with layout information elements. The advantage is that we can preserve possible semantics that users puts into layouts. A user might arrange graphical representations in a strategic way, or use empty lines in textual representation to express something *between the lines*. The second solution has complementary advantages and disadvantages.

For the purpose of embedded textual editors we propose the second solution, the automatic generation of white-spaces, for the following three reasons. Firstly, the edited text usually only comprises text pieces; white-spaces with hidden semantics, like empty lines, are not that important. Secondly, it is technically difficult: the embedded text editors rely on the modelling facilities of the host editor; storing information beyond the model is difficult or at least requires to change the host editor. Thirdly, with automatic layout, it is also possible to textually represent models that where 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.

Auto-layouting models. The automatic layout of textual representations requires additional information about the textual notation. Language independent textual layouts, like separating each token with a space, are obviously not adequate. The language notation must contain clues about whitespaces between two particular tokens.

We propose a technique that uses *whitespace roles*. Language engineers add whitespace roles as symbols to grammar rules. A whitespace role determines the *role* that the separation between two tokens plays. This conveys the semantics of the symbol separation. Whitespaces roles are dynamically instantiated with actual whitespaces, when text is created from a model. During this instantiation process, the whitespaces are created based on the given roles.

To decide, which white-space is instantiated from which white-space role, we use a software component called *layout manager*. The layout-manager creates whitespaces for whitespace roles in the order they appear within the created textual model representation. The layout manager decides with whitespace is used to instantiate a whitespace role, depending on the context the whitespace role is used in.

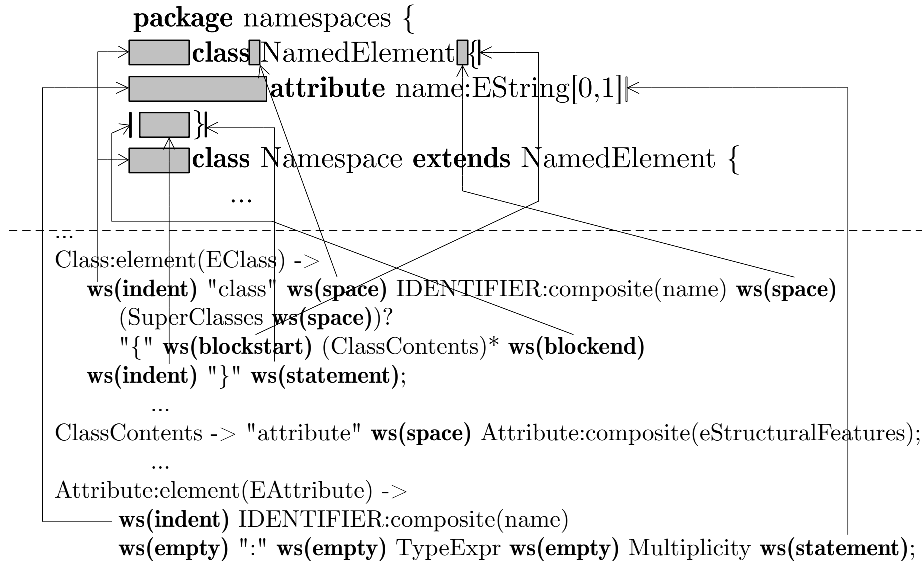

Fig. 5. An example text with withespaces for an example notation with whitespace roles.

An example: A layout manager for creating block-layouts as they are used in most programming languages, supports the whitespace roles *space*, *emtpy*, *statement*, *blockstart*, *blockend*, *indent*. The default behaviour of this manager is to instantiate each *space* with a space and each *empty* with an empty string. But, if the layout manager detects that a single line of code becomes to long to be readable, the layout manager can also instatiate both roles with a return and a followed proper identation. The layout 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.

Figure 5 shows an example model and language notation with whitespace roles for the ecore language based on a block-layout manager. Instantiated whitespaces and their whitespace roles are emphasised in the example.

Layout managers define sets of whitespace roles and define a behaviour that determines how to instantiate these roles. A framework that realises the development of textual model editors can predefine different layout managers and, or allow language engineers to define their own layout managers. Possible example layout manager can be used for the mentioned block layouts, expression languages, languages that present information in tabulars, etc.

3.3 Committing model changes

Problems caused by a different editing paradigms: Opposite to the model view controller pattern, background parsing does not change the edited model element, but creates a new ones. This causes two problems. Firstly, other model

elements, 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. Secondly, the edited model element might contain information that is not represented in its textual representation; this information will be lost, since 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 change the original edited model element and not replace it. This would solve both problems.

Anyway, both solutions require to access a model element's identity without relying on the object's identity that represents the model element in computer memory. We need to tell whether two different model elements are meant to be the same model elements. This is obviously language specific, and therefore the language's concept of identity has to be defined. For simple language constructs, this could simply be the name of a model element, stored in an according name attribute. In more complex languages, this might involve namespaces, scopes, name hiding, or qualified names.

Once identity is clearly defined, and we can tell whether two model elements have the same identity, realising the first problem solution becomes very easy. Take all references into the original edited model element, determine the identity of the referenced model elements within the original editor model element, search for an model element with the same identity within the newly created model element, and reassign 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 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 [**?**, ?,?]

One problem remains: this problem occurs, if the user changes the text representation in a way that would change attributes responsible to determine a model element's identity. Using the background parsing strategy, it is not clear what the users intentions are. Did the user wanted to change the name of a model element, or did he wanted to actually replace an 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 cause by different redo/undo paradigms: A convenient feature of model editors is the possibility to undo and redo model changes. This needs to be preserve for model changes in embedded text editors. When changing a model

in a graphical editor (or other model-view-controller editors), user actions cause commands that change the model. These commands are saved in a stack, which is utilize to undo/redo (pop/push) commands.

When using a textual model editor based on background parsing, users change a string representation of the model. 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 a completely different undo/redo paradigm.

We propose the following solution. We stick with the string based replacements during textual editing. The textual editing process will not change the host editor's model. Only when the user submits the textual changes, i.e. closes the embedded editor, the model is changed. At this point, every model change necessary to replace the original edited model element is encapsulated into a single command, which is than 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 ([5]) textual editing framework for the eclipse platform called Textual Editing Framework (TEF) [6]. 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 [7], outline view, occurrences, smart navigation, etc. An example notation description is shown in figure 2.

We integrated the development of embedded editors into TEF. Embedded editors can be created for EMF generated tree-based editors and editors created with the Graphical Modelling Framework (GMF) [8]. These embedded editors do not require to change the host editor. In theory, TEF should work for all EMF-based host editors. To use TEF for embedded editors, language engineers provide a notation description for the at least those language elements that will be used in embedded editors. 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. When an embedded editor is opened for a specific model element, the notation is subdivided into a partial notation for the according meta-model element dynamically. The editor is shown in a small overlay window attached to the edited model element in the host editor. 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 initial textual representation is created from the edited model element using layout managers and whitespace roles as defined in the notation description.

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 ESC). The later results in a compound command added to the host editors command stack. This compound command comprises sub-commands for all the model changes needed to replace the original edited model element with the edited model element created in the last background parsing. This compound command also includes sub-commands that replace references to elements in the original edited model with references to elements with the same identity in the newly created edited model element. Identity thereby either means TEF's default identity concepts or an identity concept given by the language engineer. This command executes the changes made to the model in the embedded editor. This approach allows for seamingless integrated and unlimited undo/redo.

4.2 Using TEF to Implement Example Embedded Editors

Ecore models We developed a textual notation for Ecore. This can be facilitated for embedded textual editing within the graphical Ecore GMF editor and the 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 renders the process of e.g. creating an operation with many parameters much more efficient.

The work on the Ecore editors showed a few problems. The first one is missing updates in the graphical representation after changing the underlying model with an embedded editor. Apparently, the GMF editor does not implement the model view controller pattern strictly enough, and therefore misses to note some changes in the model. We have to conclude that in some cases altering the host editor's implementation is inevitable. A second problem we already discussed in section 3: elements in the Ecore language often 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 model is edit textually. A third problem are the different identifiers in Ecore models, which should rely on nested namespaces and full qualified names. Therefore, a simply notation description for Ecore is not enough; we had to manually implement Ecore's identity concepts to improve the reference resolution in the otherwise automatically generated textual editing facitlites.

OCL integrated into other languages The OCL constraint language is often used in conjunction with languages that describe object oriented structures, or describe the behaviour of such structures. Hence, OCL expressions are often attached to the graphical notations of languages like UML, MOF, or Ecore. There-

fore editors for those languages should support OCL editing capabilities, but they usually only do so by means of basic string based text editing.

We developed an OCL editor based on the MDT OCL project. We integrated this editor with the tree-based Ecore editor. EMF validation framework requires OCL constraints stored in Ecore annotions, because Ecore itself does not support the storage of OCL constraints. Since 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 commiting the changes to OCL constraints very easy, since the embedded editor only has to replace a string annotation in the host-editors Ecore model.

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 very tiresome, if these models contain mathematical expressions. Since, mathematics is the most common means to express computation, expressions are part of many languages.

We developed a simple straight forward notation for a simple straight forward expression meta-model. This expression meta-model and notations is a blueprint and example on how easy it is to integrate sophisticated editing capabilities for expression 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

Research on creating textual notations based on meta-models is as old the first MOF standardisation efforts. Early work on grammar and meta-model relations comprises: Alanen et al [9], Scheidgen et al [10], Wimmer et al [1]. Later on, this advances into frameworks for textual model editors either based on existing meta-models (TCS [3], TCSSL [2], MontiCore [4]), or for meta-models or other parse-tree representations generated from the notations grammar (xText [11], , Safari [12]). 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 (since they are only text editors). This makes it hard to facilitate those frameworks for embedded textual editing.

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 languages through. Further attempts to describe the relations between graphical notations and textual notations have been made by Tveit et al [13].

Besides using context-free grammar and background parsing, textual modelling can also be conducted using the model-view controller pattern. This has

be done in intentional programming [14] and the meta programming system [15]. Since the editing paradigm is very close to those of graphical editors, it is thinkable that these frameworks can be integrated into graphical editors as well, maybe with more natural solutions to the identity problem. However, using the model view controller pattern, only allows to create models based on commands. This gives these editors the same limitations when it comes to editing expression structures. But, this is exactly the category of language constructs that embedded textual editing should provide a better solution for.

6 Conclusions

The work presented in this paper shows that textual model editors can be embedded into graphical editors. It also showed that this can be achieved quite practical and efficiently based on generative development using already existing technology. Even though formal studies have to be conducted on how this raises the productivity of model editors, we can conclude from the effect that language depending text editing had on the effectiveness of programming: the same effect can also be assumed for modelling/programming with graphical languages that contain structured text.

One major problem with embedded text editing based on background parsing that could not be solved quit satifactorially, is the fact that background parsing makes it hard to read the users intention, and that changes made to an objects identity might result in unwanted effects. This could be solved in two ways: one would mean to abandon background parsing and turn towards model-view controller based textual model editing, the other can be to provide refactoring capabilities to text editors, thus extend the means for the user to express his intentions.

Other future work would be to explore different grades of notation and editor integration. In this paper, we suggested to provide full notation descriptions for both the graphical and the textual notation, which allows to use both editors, host and embedded, on all model elements. But, in many cases it would be more natural to integrate notation description into one notation description with graphical and textual parts. This would require more complicated integrated description languages and framework, but would allow for more concise, coherent descriptions.

On the technical site, we could explore other possibilities to integrate editors into each other. For example, it would be desirable to have the textual editor widget direct imposed into the according graphical widget. This makes additional overlay windows superfluously and would create a more fluid editing experience. Further more, would this allow to see syntactical highlighting and error annotations directly in the graphical model view.

References

1. Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: Satellite Events at the MoDELS 2005 Conference. (2006) 159–168

2. 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: Proceedings of the 9th International Conference, MoDELS 2006. (2006) pp. 98–110
3. Jouault, F., Bézivin, J., Kurtev, I.: Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering. In: GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering, New York, NY, USA, ACM Press (2006) 249–254
4. 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. Volume 4735 of Lecture Notes in Computer Science., Springer (2007) 286–300
5. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework (The Eclipse Series). Addison-Wesley Professional (August 2003)
6. Homepage: Textual Editing Framework (TEF) See `http://www.informatik.hu-berlin.de/sam/meta-tools/tef`.
7. Scheidgen, M.: Integrating content-assist into textual model editors. In: Modellierung 2008, LNI (2008)
8. Homepage: Graphical Modelling Framework (GMF) See `http://www.eclipse.org/gmf/`.
9. Alanen, M., Porres, I.: A Relation between Context-Free Grammars and Meta Object Facility Metamodels. Technical report, TUCS (2004)
10. Fischer, J., Piefel, M., Scheidgen, M.: A metamodel for sdl-2000 in the context of metamodelling ulf. In Amyot, D., Williams, A.W., eds.: SAM. Volume 3319 of Lecture Notes in Computer Science., Springer (2004) 208–223
11. Homepage: openArchitectureWare See `http://www.openarchitectureware.org`.
12. Charles, P., Dolby, J., Fuhrer, R.M., Stanley M. Sutton, J., Vaziri, M.: Safari: a meta-tooling framework for generating language-specific ide's. In: OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM (2006) 722–723
13. Prinz, A., Scheidgen, M., Tveit, M.S.: A model-based standard for sdl. In Gaudin, E., Najm, E., Reed, R., eds.: SDL Forum. Volume 4745 of Lecture Notes in Computer Science., Springer (2007) 1–18
14. Simonyi, C.: The death of computer languages, the birth of Intentional Programming. Technical report, Microsoft Research (1995)
15. Dmitriev, S.: Language Oriented Programming: The Next Programming Paradigm. onBoard (1) (November 2004)