Preserving Non-essential Information Related to the Presentation of a Language Instance

Terje Gjøsæter and Andreas Prinz

Faculty of Engineering and Science, University of Agder
Serviceboks 509, NO-4898 Grimstad, Norway
{terje.gjosater, andreas.prinz}@uia.no

Abstract
In this article, we look at issues related to the preservation and re-generation of non-essential aspects like style and formatting for diagrams and code for different presentations of a computer language. We will primarily examine these issues from a meta-modelling perspective. We may encounter these issues when we have different presentations for a language and have to generate code/diagram fragments as a result of synchronisation between the different presentations. Similar issues may arise if we want to store the model (an instance of the meta-model that defines the language) instead of the source text or diagram, and have to re-generate the code/diagram. We describe how existing tools and approaches handle these issues, then we look at what solutions for storing the style information are possible. We develop some requirements and look at how the possible solutions support these requirements, and finally we discuss some ideas for implementation based on the recommended solutions.

1 Introduction

Domain Specific Languages (DSLs) are becoming increasingly popular, and there are some issues that may arise when we want to preserve or re-generate information regarding style and formatting of the presentation (concrete syntax) of the language. DSLs that are created based on meta-modelling technologies, are usually designed with a starting point in the structure (abstract syntax), rather than in the presentation. The same is true more and more often when traditional grammar-based compiler technologies are used. Then exchange formats and/or different presentations are added to the structure. Information on style and layout are normally not available in the structure meta-model. This *extra information* may be lost if we do not take care to preserve it for a language instance.

We will look at elements of a language that can vary, but do not carry any semantic meaning, first in textual presentation, and then in graphical presentation.

In an editor for a textual programming language, the text may vary in terms of font, size and style. However, these aspects of the text normally carry no meaning and are purely intended to make the text easy to work with for the programmer. In many

This paper was presented at the NIK-2009 conference; see http://www.nik.no/.

programming languages, white-space also carry no meaning, and can be used freely for separation of elements. Areas of variation may be indentation style (tabs vs. a given number of spaces), placement of braces, extra line breaks to visually separate sections.

Here are two different ways to express the same code fragment of the C language:

```
if (x==1)
{
   y = x;
}
```

A more compact version:

```
if (x==1) {
   y = x;
}
```

Or even more compact:

```
if (x==1) {y = x;}
```

For several languages, there are different style guidelines that put constraints on the layout and style of a language instance, however, there may still be some room for variation within a given style. For C/C++ for example, there are the GNU style, the BSD style, K&R style, Linux kernel style and several others.

Many languages also allows different ways to express the same language construct. For example in Java, we may define imports to allow the use of short names of classes, methods and variables, or we may instead choose to use fully qualified names. Constructs like this are sometimes called *syntactic sugar*, when they allow for a simpler way to express something, but it does not reduce the expressiveness of the language if the feature is removed.

Identifiers (i.e. names of classes, methods and variables) in a language carry meaning in that they indicate identity between elements, but on the other hand, the choice of name is partly a stylistic issue, and may be changed without changing the meaning of a language instance as long as all instances of a given identifier are renamed. Several different identifier naming conventions exist, so the same variable may for example be called *date*, *dateOfToday*, or *date_of_today*. The identifier names are normally preserved in the model, so re-generation may not be an issue, although changing them in order to make them conform to naming conventions may be an option.

In a graphical presentation, there are several possibilities for variation that do not carry semantic meaning. Placement (and implied grouping) of elements in a diagram is the biggest issue in the graphical presentation, but we also have possibilities for variation in the style of lines for aggregation, association and inheritance; are the lines straight (see Figure 1), with angles (see Figure 2), or curved? How wide is the line and how big are the arrows? As in a textual presentation, text within the diagram may also vary in terms of font, size and style.

We will primarily look at these issues from a meta-modelling perspective. That means that most of the examples we provide, the tools that we examine, as well as the solution we propose, relate to that perspective.

The rest of the article is organised as follows: In Section 2 we will give some examples to illustrate the issues at hand, in Section 3 we will give a brief overview of our view of

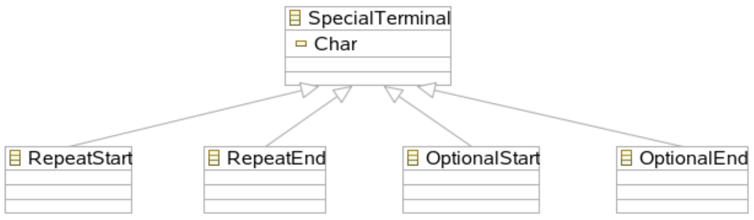


Figure 1: Class diagram, with straight diagonal lines

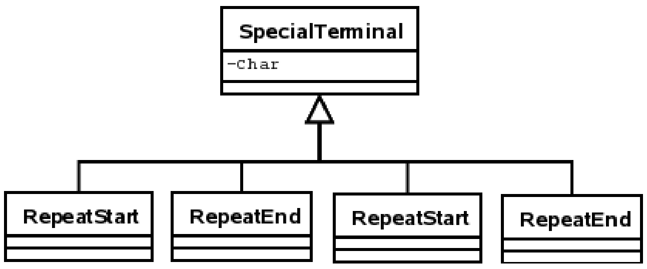


Figure 2: Class diagram, with angular parallel lines

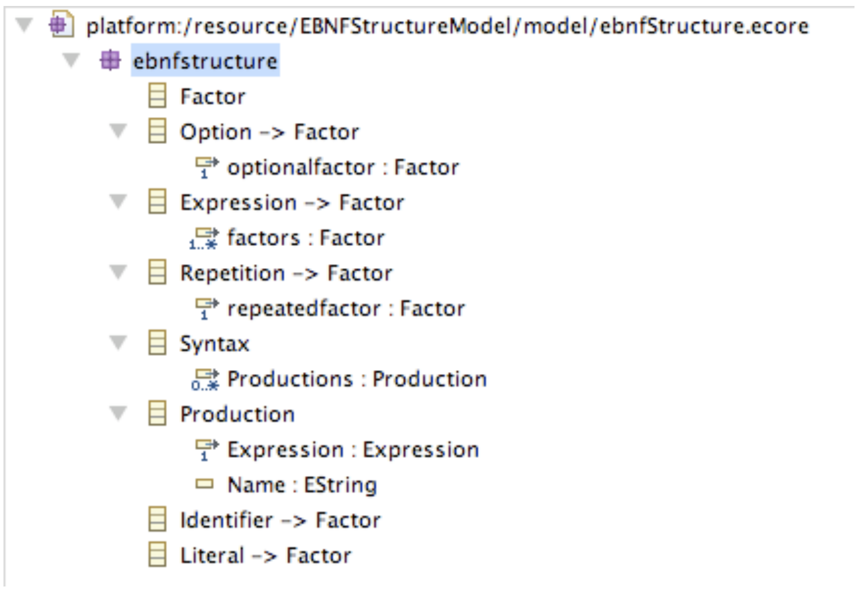
the different aspects of a language. How existing editors handle storage of style and layout information will be described in Section 4, and in Section 5 we will develop some requirements for features we want from editors, and we will examine what options are available to implement them. Finally, we draw some conclusions in Section 6.

2 Examples

In this section, we will describe some common cases where these issues arise, and derive requirements from them.

One issue may show up when we want to have multiple presentations of a language. It is not uncommon to have different presentations of the same language, for example one textual and one graphical presentation, as is the case for SDL (Specification and Description Language, see [1]). Another example is the Eclipse-based meta-language Ecore (see [2]), that can be edited both in a tree editor and in a graphical editor. This gives the first requirement: *R1 - Editors should support different presentations and different views on the same presentation for the same language instance.*

If we want to edit different aspects of a language instance simultaneously via different presentations, the issue will arise about how to synchronise and update the different presentations while preferably inserting the generated elements in a way that it will fit with the existing style of each presentation. EMF (the Eclipse Modelling Framework) uses the MVC pattern (Model-View-Controller, see [3]) to support synchronisation between different editor views on Ecore models. When a file is saved, the other views are automatically updated. If an element is added in one view, it will appear in the other view after a save. In the GMF-based graphical Ecore editor, the automatic layout engine will then attempt to find a suitable location for new elements that are generated because of synchronisation. Whenever there is some room for variation in the layout and style of a language instance, fragments of generated code or diagram may or may not fit well with the layout and style of the existing code or diagram, depending on the code generation or automatic diagram layout mechanisms in each editor. In some cases, like with the tree view of an Ecore diagram as shown in Figure 3, there is little or no significant room for variation, while in a graphical editor, the same model can be laid out in many different ways. From this, we derive the second requirement: *R2 - Editors should support generating fragments in an auto-detected or explicitly chosen style.*



```
platform:/resource/EBNFStructureModel/model/ebnfStructure.ecore
  ebnfstructure
    Factor
    Option -> Factor
      optionalfactor : Factor
    Expression -> Factor
      factors : Factor
    Repetition -> Factor
      repeatedfactor : Factor
    Syntax
      Productions : Production
    Production
      Expression : Expression
      Name : EString
    Identifier -> Factor
    Literal -> Factor
```

Figure 3: Tree-view of an Ecore model

It is sometimes useful for the modeller to keep related elements together. In Figure 4, we see that *Identifier* and *Literal* are manually grouped together, as well as *Repetition* and *Option*. In the automatic layout shown in Figure 5, this grouping is not preserved. Although the automatic layout is not always optimal, it is still necessary to have some way to render generated elements, so this gives us the third requirement: *R3 - Graphical editors should support automatic layout of diagrams.*

When storing a language instance, the developer of meta-model based textual editors faces the choice of saving the text, or to save the model and re-generate a default form of the text from this model. Alternatively, the model could be saved together with some form of presentation of style and layout elements. The latter option is commonly chosen for diagrams, for example in the XMI format (XML Metadata Interchange, see [4]) that accommodates both model structure and (optionally) layout information. The fourth requirement covers this: *R4 - The language instance should be preserved in such a way that style and layout information is preserved.*

The issues related to style and layout may not only be about aesthetics and readability; in some cases it may lead to problems if we do not take these issues into account. If several programmers are working on the same source file, there may be problems related to the configuration of the editors used by the programmers; for example choice of character encoding for strings and comments, and indentation style. In languages like Python, where indentation is used as block delimiters and therefore must be consistent, mixed indentation styles in the same file may lead to errors. From this, we get the fifth requirement: *R5 - Editors should support standardised user-definable and user-selectable styles.*

In the following section, we will give a brief overview of our view of the different

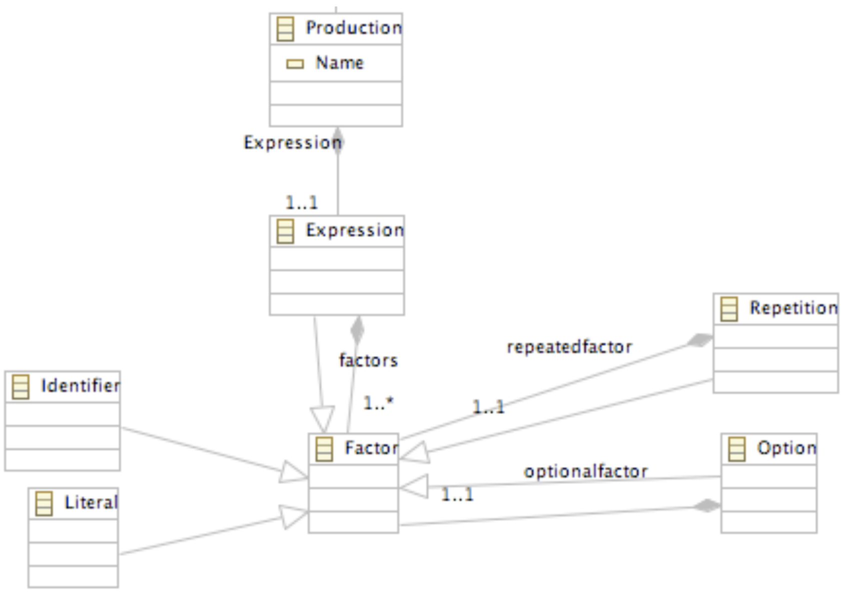


Figure 4: Graphical view of an Ecore model, with manual layout

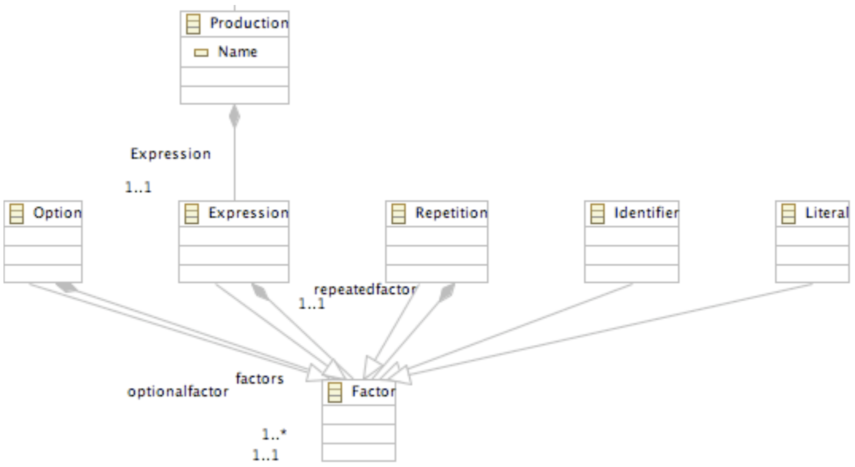


Figure 5: Graphical view of an Ecore model, with automatic layout

aspects of a computer language, in order to give some background for understanding the issues and proposed solutions.

3 Aspects of a Language Description

Figure 6 shows the aspects of a complete description of a modelling language, consisting of structure, constraints, presentation and behaviour. This view was first introduced in [5].

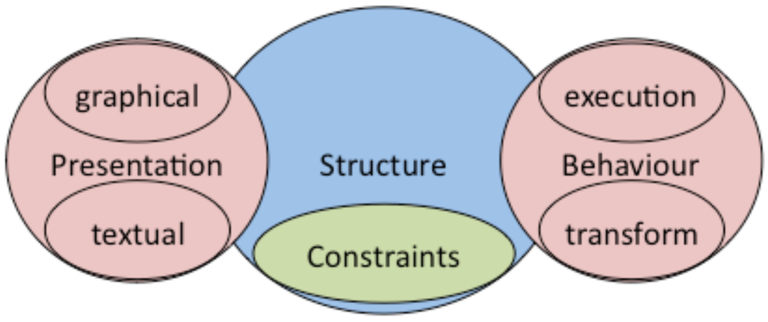


Figure 6: The aspects of a modelling language

Structure defines the constructs of a language and how they are related. This is also known as abstract syntax.

Constraints describe additional restrictions on the structure of the language, beyond what is feasible to express in the structure itself.

Presentation defines how instances of the language are represented. This can be the description of a graphical or textual concrete language syntax.

Behaviour explains the dynamic semantics of the language. This can be a mapping or a transformation into another language (translational semantics, denotational semantics), or it defines the execution of language instances (operational semantics).

As Figure 6 shows, the structure, including constraints, is the central aspect in a language description, and all the language aspects are defined completely independently with clear boundaries between them.

The presentation of a language describes the possible forms of a statement of the language. For a textual language, the presentation describes what words are allowed to be used in the language, what words have a special meaning and are reserved, and what words are possible to use for variable names. It may also describe what sequence the elements of the language may occur in; the syntactic features of the language. For a graphical language, the presentation will express what different symbols are used in the language, what they look like, and how they can be connected and modified to form a meaningful unit in the language.

Elements of text and diagrams that may vary, but are not explicitly represented in the structure meta-model, and in some cases not even in the presentation meta-model, are our main concern in this article. In the following section, we will look into how we can handle and preserve these elements.

4 Storage Options

In this section, we examine what are the possible ways to handle the issues at hand, and how existing tools and technologies handle some of the issues that we are concerned with. How do we capture the extra information about style and layout? There are some different approaches to handling storage of a language instance:

Complete concrete form (e.g. source code): In this case we have to parse code or diagram again, but all information except editor defaults for style and layout is stored.

Presentation model: We save the language instance as an instance of a presentation meta-model. If the presentation meta-model also supports white space, comments, layout information and other style information, this is also adequate for preserving all or most of the relevant extra information. Note that textual languages sometimes do not have an explicitly defined presentation meta-model but instead use a textual grammar to define the presentation, with a mapping to the structure meta-model.

Structure model: In this case, we store an instance of the structure meta-model. This is more abstract than the previous option, and information related to style and layout is lost, and some "syntactic sugar" may also be lost. When this is used as a storage format, we have to generate a default presentation model or directly generate code/diagram in a normalised form for editing.

Some languages have stricter requirements about style and layout, for example the Python programming language, which enforces a specific white-space style, thereby avoiding some of the issues of preservation of style.

We also note that there is software like GNU Indent (see [6]) that can reformat C and C++ code to conform to a given style. Other code beautifiers/prettifiers/pretty printers do something similar for a range of textual languages.

It is interesting to note that in the case of HTML (Hyper-Text Markup Language, see [7]), it is recommended to use external CSS (Cascading Style Sheets, see [8]) files to define the style of a web page, thereby separating style and content. This allows us to create different views on the same content, and this is clearly a useful feature, particularly when it comes to having different styles and layouts for different media like printers, normal sized screen and small screen. It also allows the user to override the default style with his own style sheet conforming to his own special preferences and requirements.

If we consider the possibility to have external style definitions, we get additional options:

Structure model with extra style and layout information: It is possible to keep extra style information separate or in the same file. In the case of textual languages, simply storing the code in addition to the model is also a simple way to preserve elements that are not represented in the model.

There are two relevant approaches to storing style and layout information:

Ad-hoc style preservation: The style information from the presentation model is explicitly preserved to supplement the structure model. When a presentation model is generated from the structure model, ad-hoc style information is used for the relevant stylistic options. In this way, with the structure model and the ad-hoc information, we are able to re-generate the original presentation model.

Default style definition: We have a default style definition that covers all stylistic options that are needed for generation of a presentation model from a structure model. When ad-hoc style information for a given stylistic feature is not preserved with the storage model, a re-generation of text or diagram will occur with the selected default style enforced. In this way, with a structure model and a default style definition, we are able to re-generate a normalised form of the original presentation model.

Goldschmidt, Becker and Uhl (see [9]) have examined how 11 different tools and approaches for textual presentation handle storage of an instance of textual presentation. It turns out that most common, 6 cases, is to store both the model and the text. There are 3 cases of storing only the text, while there is only one case of storing only the model without text. Finally there is one case where it depends on the implementation of the approach. When only the model is stored, the text has to be re-generated when the editor is re-started. Any information from the original text that is not preserved in the model will then be lost.

For graphical meta-model-based tools, a common approach is to save the model and layout information as a single XMI file, or to keep the model in one XMI file and refer to that from a second XMI file that contains only layout and style information. This is the case with editors generated with the GMF (Graphical Modelling Framework, see [10]). One example is the GMF-based Ecore diagram editor; the Ecore file format is XMI and the Ecore_diagram format is XMI too, but the latter contains just style and layout information and refers to the former for model information.

XMI is able to store any MOF-based (Meta-Object Facility, see [11]) model, including style and layout information. Although XMI is a very useful standard for storing models in a cross-platform format, the different incompatible dialects of XMI make transfer between different implementations difficult.

5 Wanted Features and Implementation Ideas

In this section, we will look at what features we would like to see in tools and technologies for editing language presentation instances, and some ideas about how these features could be implemented.

Requirements

From 2 we have the following requirements for features that would be desirable in tools and technologies for editing language instances:

1. Support a) different presentations and b) different views on the same presentation for the same language instance.

2. Generate fragments in a) auto-detected or b) explicitly chosen style.

3. Automatic layout of diagrams.

4. Preservation of ad-hoc style information.

5. Standardised user-definable styles/layouts.

Table 1: Storage approaches versus requirements

	1a	1b	2a	2b	3	4	5
Source	+	+	+	-	+	+	-
Source with style definitions	+	+	+	+	+	+	+
Presentation model	+	+	+	-	+	+	-
Structure model	-	+	-	-	+	-	-
Model with ad-hoc style	+	+	+	-	+	+	-
Model with default style definitions	+	+	-	+	+	-	+
Model with ad-hoc and default style definitions	+	+	+	+	+	+	+

From Table 1, we see that only two of the storage options support all our requirements; *Source/diagram with default style definitions*, and *Model with ad-hoc and default style definitions*. In the following section, we will discuss some issues related to implementation of these two options.

Implementation

In this part, we will give some brief ideas on how it is possible to implement the requirements listed above.

Separation of content and style

Separation of content and style facilitates different views on the same content. Both recommended storage options, *Source/diagram with default style definitions* and *Model with ad-hoc and default style definitions*, depend on separation of content and style. We need a standardised system for defining styles independent of content, similar in principle to CSS for HTML.

For the pure form of the content of a language instance, we find it in the most abstract form in the structure model. If we want to store the structure model and still be able to re-generate the original form of the presentation of a language instance, we have to store the structure model together with either the presentation model or with ad-hoc style information.

For textual presentations, a good option would be to save the structure model, and in addition, saving the text if ad-hoc style is supported, and finally to store a reference to a default style definition if it exists.

For diagrams, a good option would be to store the structure model, and additionally a file containing ad-hoc style information and (optionally) a reference to a default style definition. From the structure model and the ad-hoc style and layout information, the diagram can easily be fully re-created.

Storing ad-hoc style

It is possible to store style and layout information that is not preserved in the model, when additionally storing the full text, or in other manners saving all style information in the case of diagrams.

In the case of textual presentation, when we save the text itself, it is easy to parse to re-create the model, and it contains exactly what we want to preserve when we want to keep the ad-hoc style information.

For graphical presentation, storing the structure model as XMI, and in addition storing layout and style information in form of an external XMI file referring to the model, will not only allow us to preserve the style information, but will also allow for different views of the model. This may be particularly useful for large models where we may want to highlight different parts of the model for different purposes.

Standard formats for textual and graphical default style definitions

Standardised, customisable and extensible style definition formats for textual and graphical presentations would be highly desirable.

The editor should supply an option for choosing between existing style definitions. The editor should be able to warn when a style rule is broken, (optionally) auto-correct the input and convert between styles. The user should be able to define and apply his own styles. A style editor would be particularly helpful for diagram styles.

For the style definition, a simple option is to supply a rule for every variable stylistic element, i.e. for every white-space or every diagram element type. For textual presentations, this could have the form of textual rules for each possible element. In the same way as XMI can be used for storing graphical stylistic and layout information, it can also be used for storing default style definitions for a given language presentation by storing style information not for existing elements, but for all possible types of elements.

Another option would be to develop an “extra-information” meta-model as an extension to each presentation meta-model. This meta-model could be instantiated to contain either ad-hoc style information or a default style.

It would also be useful to be able to define rules for diagram element placement, i.e. to have an editor-independent way to define different layout strategies.

Auto-detection of style

How is it possible to analyse the style of existing text and diagrams? We may need a separate “extra-information” meta-model that covers all stylistic elements of a given language presentation. From this, a language instance can be analysed. If the language instance is consistent and conformant in choice of stylistic options, it should be trivial to detect the style, however, we also need to handle deviations from standard styles and internal inconsistencies. An option would be to implement some metrics to calculate the closest available style. The user should be informed about the detected style, and be given the choice to accept it or chose another available style, as well as given the choice if the selected style should be strictly enforced or not. It could also be possible for the user to chose to generate a style definition from the existing language instance.

Generate fragments in auto-detected or explicitly chosen style

When a fragment of code or diagram in a given presentation is generated, it should be conforming with the existing style for that presentation instance. When a style is auto-detected or explicitly chosen, it is trivial to apply this style to generated code fragments or diagram elements. However, for graphics, layout of generated elements requires an "intelligent" layout engine, and this is non-trivial to do in a fully satisfactory way since developers like to place and group items in various meaningful ways, and automatic layout may not always be good enough.

Synchronisation
Elements that are not represented in the structure instance, will not automatically be preserved if synchronisation between two presentation instances happens via the structure with no direct interaction between the different presentation instances, thus stylistic information and some syntactic sugar may be lost in the abstraction that happens when structure model elements are generated from a diagram or text. By storing the ad-hoc style information, this extra information can be preserved.

It is important to consider if we want to generate syntactic sugar that is not represented in the structure meta-model, i.e. do we aim at generating constructs with the highest possible level of abstraction, or not. If two presentation meta-models contain common constructs that are not represented in the structure meta-model, it is important to handle this issue by storing information about the constructs in the ad-hoc style information so these constructs do not get lost from one presentation instance to another.

6 Conclusions

It is common to have different alternative presentations of a given language, and we want to support synchronisation between the different presentations of the same language instance in different editors. When the synchronisation is performed via the structure of the language, information that is represented in the presentation meta-model but not in the structure meta-model, is lost unless it is explicitly preserved.

When we want to store the structure model of a language instance rather than the original source, we lose information that is not represented in the structure meta-model when we re-generate of the language instance for editing.

These issues can be handled by storing style and layout information from the language instance in addition to the structure model. Additionally, it would be useful to have default style definitions that apply to generated code and diagram fragments.

In the case of textual presentation, storing text, and using external tools for reformatting according to style guidelines, may satisfy most of our requirements. However, a more elegant approach is to use external style definitions, that editors may optionally enforce in editing, use for applying style to generated code, and use for conversion between registered styles.

Preservation of style and layout as well as user-definable external style definitions would be the best choice in order to allow the user/developer to have different independent views on the same language instance. The existing XMI format supports this, as seen with GMF-based editors where style and layout information is external to the model. However, the issue of quality and usability of automatic layout versus manual layout of elements in a diagram remains an area for further research.

References

[1] ITU-T: SDL - ITU-T Specification and Description Language (SDL-2000). ITU-T Recommendation Z.100 (1999)

[2] Griffin, C.: Using EMF. Technical report, IBM: Eclipse Corner Article (2003) See also `http://www.eclipse.org/articles/Article-Using EMF/using-emf.html`.

[3] Reenskaug, T.M.H.: Thing-model-view-editor an example from a planningsystem. Technical report, Xerox PARC (May 1979)

[4] OMG: MOF 2.0/XMI Mapping Specification, v2.1 formal/05-09-01. OMG document, Object Management Group (2005) Available at `http://www.omg.org/docs/formal/05-09-01.pdf`.

[5] Nytun, J.P., Prinz, A., Tveit, M.S.: Automatic generation of modelling tools. In Rensink, A., Warmer, J., eds.: ECMDA-FA. Volume 4066 of Lecture Notes in Computer Science., Springer (2006) 268–283

[6] GNU indent developers: GNU indent manual. GNU, http://www.gnu.org/software/indent/manual/. 2.2.10 edn. (July 2008)

[7] Raggett, D., Hors, A.L., Jacobs, I.: HTML 4.01 Specification. Technical report, W3C, http://www.w3.org/TR/html401/ (December 1999)

[8] Bos, B., Çelik, T., Hickson, I., Lie, H.W.: Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. Technical report, W3C, http://www.w3.org/TR/CSS2/ (April 2009)

[9] Goldschmidt, T., Becker, S., Uhl, A.: Classification of concrete textual syntax mapping approaches. In: ECMDA-FA 2008. Volume 5095 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2008) 169–184

[10] GMF developers: Eclipse Graphical Modeling Framework. (2008) See `http://www.eclipse.org/gmf`.

[11] OMG Editor: Revised Submission to OMG RFP ad/2003-04-07: Meta Object Facility (MOF) 2.0 Core Proposal. Technical report, Object Management Group (April 2003) Available at `http://www.omg.org/docs/formal/06-01-01.pdf`.