

# Primary Storage Formats for Textual DSLs

Peter Friese  
itemis AG  
Schauenburgerstraße 116  
24118 Kiel, Germany  
peter.friese@itemis.de

Jan Köhnlein  
itemis AG  
Schauenburgerstraße 116  
24118 Kiel, Germany  
jan.koehnlein@itemis.de

Heiko Behrens  
itemis AG  
Schauenburgerstraße 116  
24118 Kiel, Germany  
heiko.behrens@itemis.de

Sven Efftinge  
itemis AG  
Schauenburgerstraße 116  
24118 Kiel, Germany  
sven.efftinge@itemis.de

Sebastian Zarnekow  
itemis AG  
Schauenburgerstraße 116  
24118 Kiel, Germany  
zarnekow@itemis.de

Dennis Hübner  
itemis AG  
Schauenburgerstraße 116  
24118 Kiel, Germany  
dennis.huebner@itemis.de

## ABSTRACT

External textual DSLs receive a lot of interest these days as they apparently solve many problems in model driven software development. Looking at the various user interaction patterns, it seems that text seems to be the best fit for most of these patterns except for searching. Model repositories are often seen as a solution for this challenge. This position paper discusses various approaches for model storage in different kinds of repositories and what their relative merits are.

## 1. INTRODUCTION

External textual DSLs receive a lot of interest these days as they apparently solve many problems in areas where more graphical-oriented modeling solutions couldn't convince in the past.

Using text as primary format for models is great for many developers as they are used to working with textual data. Developers can leverage sophisticated tools around the manipulation of text and a stream of characters is a simple structure that is widely understood. Also, text integrates well with most tools used in today's software development such as revision control systems, build servers, etc.

However, textual models do not lend themselves to being searched so well, which - among others - is the major reason why people dealing with large models are seeking to use an advanced approach of storing data. Model repositories seem to provide a better way of analyzing data in that manner.

This position paper aims at adding to the discussion by the following approach: first, we discuss common user interaction patterns that apply when working with textual DSLs. After that, we elaborate on various approaches to realize

model storage for textual models. As not all of the approaches have been implemented and tested in real life, the paper concludes with a list of possible next steps.

This discussion concentrates on DSL users rather than DSL developers.

## 2. INTERACTION PATTERNS FOR TEXTUAL DSLS

Text has been the prevalent representation for information in computer sciences ever since, resulting in an enormous wealth of tools that are at a developers disposal. Working with text feels so natural to most developers that one rarely realizes the various interaction patterns that are involved when dealing with text.

Among other things, textual DSLs aim at increasing developer productivity. It is thus essential to ensure common interaction patterns for text still work for textual DSLs. As we will see later in this paper, the choice of a storage representation has considerable impact on the availability of certain services.

Relevant services for textual DSLs comprise the following:

*Editing:* even though editing is the most basic operation applied to text, it is important to point out that operations such as deleting and inserting text as well as navigating the text document with the caret need to be supported in a natural way to ensure a genuine user experience.

*Navigation:* textual models need to be navigable both on a local and a global level. Local navigation can be facilitated by providing navigation views such as (hierarchical) outlines and structure-aware keybindings (e.g. CTRL+Down navigates to the next block structure). To allow for global navigation of textual models, DSL toolkits might choose to provide hierarchical outline views that show the entire model. As the number of model elements can quickly grow into dimensions which make such outlines unusable, lookup dialogs might turn out to be an alternative means of navigation, especially for users who know what they are looking for. Most textual models contain cross references between model elements, which calls for navigation support in this area as

well: as these cross references can be thought of links between the various parts of a model, hyperlink navigation - well-known from web browsers - comes as a natural fit for this challenge.

*Searching:* the aforementioned lookup dialogs can be seen as a specialized interface into a search service, and in fact both are likely to use the same backend infrastructure: to ensure a responsive UI, search facilities often rely on an index. However, indexes can usually only contain elements that are identifiable. Information in non-identifying model attributes will be ignored, unless a full-text index is provided. Thus, most text editors feature full-text search facilities that operate on plain text.

*Refactoring:* as models grow larger, developers inevitably need to restructure and recompose them, a process commonly known as refactoring. To ensure efficient use, DSL tooling must provide elaborate refactoring capabilities, lest developers end up in search and replace nightmares. However, the amount of refactoring (beyond rename) is very language / domain specific. Some languages may not need any elaborate refactoring needs at all. Expression languages are an example for this kind of language.

*Versioning:* models and other source code files are the assets of every software development undertaking. They need to be safely stored, ideally in one common location. To allow for a controlled software development process, it is essential that these assets be versioned at the various stages of their lifecycle. Source code repositories (SCRs) like CVS and Subversion (SVN) provide suitable services to achieve these goals. However, most SCRs are geared towards text files. Although storing binary files such as images and (UML) models are possible, performing operations like diff and merge cannot be performed on binary files due to their very nature. Tool vendors have been known to circumvent such problems by providing specialized "Team Servers" that act as a model repository. Unfortunately, these approaches do not provide a reliable solution for MDS environments in which it is vital to be able to create baselines for models and the derived artifacts in an atomic way.

*Collaborative work:* often, it is desirable to extract certain parts of a model, e.g. for sending it to a co-worker or pasting it to a discussion forum. Recipients of these model fragments should then be able to import them into their model workspace in order to continue working on them. Both exporting and importing require that either the model files are stored in a textual representation, or the textual DSL editors support clipboard usage, or both.

*Command line tools:* as mentioned in the introduction, a vast array of tools for text processing exist. Being able to process textual models using these tools without any extra indirection clearly is a benefit.

### 3. USABILITY CONSIDERATIONS

Several interaction patterns have special usability requirements that have to be met in order to be accepted by developers. Here is a brief discussion of the most important ones:

*Editing:* Users expect normal editor behaviour when editing textual models. This is especially true if the DSL editor is tightly integrated with an IDE platform such as Eclipse or Visual Studio. In particular, most users are accustomed to being able to edit files in their entirety in an editor. Editing various parts of a file in disparate editors is quite unusual and might result in confusion.

Following behaviours might result in confusion:

- the current editor can only modify and manipulate part of a file
- an editor modifies several resources at once

*Navigation:* as already pointed out, several navigation facilities are required for efficient navigation of textual models. Depending on the size of the models being dealt with, certain facilities will be more suitable than others. Outlines are more suitable for small models or parts of a big model. Lookup dialogs can be a better choice for large models, as they inherently offer a filter mechanism. However, resource changes must be reflected by the underlying index immediately. Otherwise, usability decreases.

*Versioning:* as text lends itself to merging and diffing so well, this advantage must be preserved when working with textual DSLs. Users must not be confronted with the abstract syntax of their models, as they are likely to be confused with all the meta classes and attributes they may never even have heard of. Comparison viewers should show the compared model in the same notation as is used in the editor. For textual DSLs this means diff tools must show a textual representation, not a tree-like structure comparison whatsoever.

## 4. STRATEGIES FOR STORING MODELS

Throughout the following text, we will be using the term "repository" a lot. To make a clear distinction, we chose to use the following terms:

- a source code repository (SCR for short) is a repository that is used to store source code and other pieces of information mostly in a file-oriented fashion. Examples are CVS and Subversion (SVN).
- a model repository (MR) is a repository whose purpose is to store models. It is not intended to store source code or things other than models.

### 4.1 Store Models as Text in a SCR

Source Control Repositories (SCR) such as CVS or SVN are commonly used in today's software development projects to store data and serve as a foundation for collaboration. The available tool chain as well as established processes concentrate on simple files though and common operations like merging or inspecting differences between two versions work best with textual data. SCRs in general directly map to a file system the developer can access with any tool and can store any kind of data next to the primary artifacts. This allows developers to manipulate arbitrary even model-independent data with very simple yet effective tools. Since typical SCRs

do not provide a metamodel nor apply any semantic checks they can even store incomplete or erroneous models as a piece of work of an intermediate state.

The storage of textual models lead into disadvantages when a semantic model is needed. A transformation step is necessary each time the developer wants to scan for structural or complex semantic information. Due to the physical separation of files, large models are typically split into different chunks to keep each file manageable. To produce a "global view" on the semantic model and follow potential cross references each file has to be taken into account again. Therefore, the analysis of model elements that are not contained inside a single file might be time-consuming or even impossible for large models that cannot be instantiated as a whole on the developer's machine. The same is true for consistency checks unless you express such operations as a server-side activity. Another weak point of this approach is the lack of support for metamodel evolution and refactoring. As large models are scattered across a large number of smaller files, each of these files need to be touched in order to perform refactorings and metamodel evolution.

## 4.2 Store Models in their native representation in a Model Repository

To overcome the disadvantage of having to recreate a ad-hoc semantic model to be able to perform structural actions, storing models in their native representation in a Model Repository (MR) might be a possible solution. The MR is backed by a (possibly relational) database whose schema is defined by the metamodel (or metamodel, to allow for utmost flexibility) of the models being stored. In the Eclipse Modeling Project, CDO implements this approach. As models are now stored in a database, operations like searching and refactoring become more efficient. Providing global consistency checks also becomes feasible by implementing suitable callbacks in the MR server. However, as developers need to work with the textual representation of the models in their workspace, models need to be converted back and forth when communicating with the MR.

As models and derived source code artifacts will now be stored in disparate repositories, versioning those artifacts together becomes more or less impossible, which poses a major problem for model driven development processes. Additionally, as models now need to comply with the metamodel, storing incomplete or partially invalid models becomes impossible. As structural changes to the DSL will result in a modified metamodel, those changes inevitably will require an update of the DSL metamodel in the MR. Native tools such as command line text editors and Unix-style tools like grep, sed and awk can no longer be used in this scenario, unless some kind of intermediate layer is introduced which acts as a facade to the model repository.

## 4.3 Store Models as Text complemented with look-up data

This is a not-so-obvious, yet already existing approach. The MR will store plain text and these "models" will remain in their textual representation throughout their life cycle. When retrieving the textual representation of a stored model no conversion is required. Thus, the original textual repre-

sentation will be fully preserved. In addition to the storage of plain text the MR manages an index to allow complex semantic analyses. The MR keeps this index in sync with the stored text and rebuilds relevant parts of the index upon arrival or modification of (new) text. This approach can be found in today's file systems that are complemented by a search index. Databases themselves often distinguish between high-performance data movement and derived look-up data. Even though libraries like Lucene can keep track of changes and could manage such a generic search index autonomously the produced look-up data would be too generic without further assistance of the DSL developer. From the perspective of the MR a meaningless stream of characters again needs some sort of parsing in order to reveal its semantic information. Modern IDEs with versatile refactoring capabilities use an optimized parser and linker to build up an in-memory AST. A generic MR would need a set of strategies such that the semantic model of each DSL could be extracted and stored for future look-up. Based on the model index, services such as attribute-based search, mode validation, refactoring and metamodel evolution become feasible.

This approach combines some benefits of the native representation while keeping the complexity at a manageable level.

## 5. CONCLUSION

The key points for the success of any of the approaches are end-user usability and performance. All approaches discussed in this paper have some drawbacks. Some of them are more suited for small- to medium-scale models, others promise to be more suited for large-scale models. Without doubt, any of the approaches can be used to work with models of almost any size. However, depending on the underlying repository technology one or the other turns out to be more suitable. While textual representations seem to be more suitable with respect to end-user usability and integration with Unix-style tools such as vi, sed, awk and grep, they have significant drawbacks when it comes to handling large-scale models. These problems might be overcome by applying indexing solutions that have been used in the search engine business with great success. Storing models in a native format (i.e. more conformant to their metamodel) solves many problems that arise when dealing with large models. However, additional steps need to be taken to make such representations usable for end-users. The discussion in this paper suggests that using text as the primary storage is well-suited for most use-cases. Only large-scale models seem to gain from non-textual storage. It remains to be seen if this is actually true. In order to prove the assumptions in this paper, more in-depth research needs to be done.

## 6. ACKNOWLEDGEMENTS

Thanks to Bernd Kolb, Axel Uhl and Markus Völter for feedback on earlier versions of this paper and discussions on the topics mentioned herein.