Proxima in practice
building a slide-show editor

Gideon Smeding

January 21, 2008

Abstract

Proxima is a very new editor framework built to facilitate the implementation of many kinds of editors. This report describes the implementing of a slide-show editor using Proxima. The slide-show editor has very limited capabilities; it only serves as a case study of Proxima. With the experience gathered by the case-study, Proxima's strengths and weaknesses are discussed.

1 Introduction

Many computer applications involve some kind of editing. Editing documents, a system's configuration, or a program, all share functionality but rarely code. To facilitate the development of these editors a generic framework is under development at Utrecht University. This framework, called Proxima[3], is implemented in Haskell and the attribute grammar system AG[1].

To study Proxima's performance in practice, a minimal "what you see is what you get" slide-show editor will be implemented using the framework. The goal of this study is twofold:

1. Proxima has barely been exposed to the real world. It has just matured from an almost unusable editor into a barely usable independent framework. Currently any input as to how it should be developed further is very valuable. In that light, the case study will serve as a review of Proxima as an editor framework.

2. Since Proxima is to be used by programmers to create editors. The lack of documentation could deter possible users; currently one has to derive from the existing editor how to implement an editor. Recording the experience of writing an editor with Proxima should alleviate the burden of future programmers.

2 Proxima

Proxima, a "presentation-oriented editor for structured documents", is an editor framework developed by Martijn Schrage, described in full in his PhD thesis[3]. The editor, or rather the editor framework, is intended to be used for documents that have a hierarchical structure. Specifically XML based documents and programming languages are good candidates.

Many more generic editors have been build. What makes Proxima unique is that its powerful presentation mechanisms can be combined with a flexible edit model. In other words: editors can have both complicated, document specific presentations, and specialized edit operations. One could, for example, define a graphical equation editor that allows variable renaming on a semantic level rather than a simple find/replace.

2.1 XPrez presentation library

A combinator library was designed to handle the presentation aspect of Proxima. This library, named Xprez[4], mainly handles static presentations but has some support for GUI (graphical user interface) related concepts like mouse clicks and keyboard controlled focus. It has primitives such as rows, columns, lines, boxes, text, and font controls. The implementation is based on a Haskell-GTK interface called gtk2hs.

2.2 Code generation

The Proxima framework also includes a generator, which prevents the user (a programmer) from having to write 'boilerplate' code. It generates type specific code that is very similar for different editors, but would otherwise have to be written by hand.

The generator reads a Haskell-like data definition and generates (among other things):

- Haskell and AG data definitions extended to account for parse errors and the like
- commonly used functionality like an XML parser and XML-based presentation.
- various class instantiations and utility functions for the generated types

2.3 Architectural overview

Proxima is a strongly evolving software package, because of this the architecture as described by the thesis, is not always apparent in the code. The underlying ideas however, are clearly there. The thesis also has a more theoretical focus on the structure. It emphasizes the flow of information as the document AST (abstract syntax tree) is transformed in stages. When programming, one naturally is more interested in the mappings that implement the transformations.

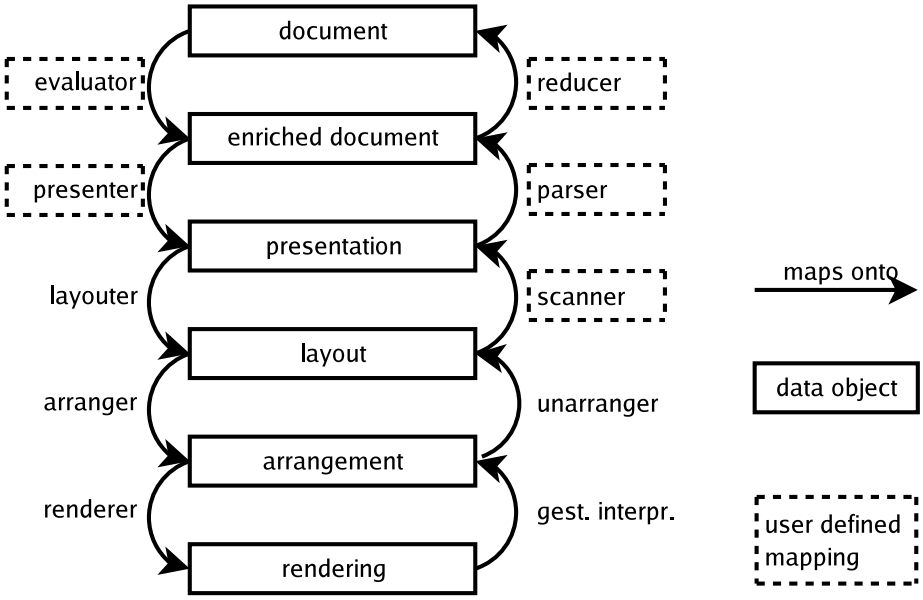

Figure 1: A flow oriented view of Proxima

The document data definitions are a very small part of the work, in comparison to the time spent on the mappings.

2.3.1 Information flow

Martijn Schrage's thesis describes the abstract information flow that is the spine of Proxima. As a rationale for Proxima and a complete understanding of Proxima, the information flow is crucial, therefore a short description is included here. The thesis provides a more complete explanation.

Proxima can be subdivided into layers of data mappings. Everything starts out as a document which is mapped onto an enriched document, which in turn, is mapped onto a presentation, and so on until the rendering layer is reached. At the rendering level the mappings are reversed: the rendering is mapped onto to the arrangement, and so on until the document level is reached.

Diagram 1 shows Proxima's full cycle of mappings. Each translation has a reversed version, thus forming a bidirectional mapping between two layers. The layer's names might be confusing at times, for example the parser and scanner do not operate on strings nor lists per se. Nevertheless their functionality closely resemble the traditional scanning/parsing process.

The types of data between the layers can be described as follows:

document An abstract document representation, comparable to a DTD (Document Type Definition) for XML. This represents the abstract syntax tree resulting from parsing a document file.

enriched document The abstract document representation extended with attributes, which contain for instance section numbering or an index. The enriched document may also have an ordered representation of some of the elements found in the original document.

presentation An abstract representation of the (enriched) document's visual representation. This presentation is described by an embedded domain specific language, implemented by the combinator library Xprez. Assuming a mostly textual presentation of a document, it should be clear that the mapping from an enriched document to a presentation can be compared to a pretty printer, and the reverse mapping would be like a parser.

layout A restructured presentation where tokens formatted with, for example, whitespace. Interestingly, mapping the layout onto the presentation can be considered a form of scanning. Especially when user input is involved.

arrangement A presentation where every object in the presentation has been given a size and position. The abstract positioning primitives like row and column have been replaced by exact positions.

rendering The on screen rendering of the arrangement, i.e. a bitmap of the window's or screen's contents.

2.3.2 Proxima component-wise

In the transformation diagram we already encountered 'sheets' that specify the mappings. Here those sheets will be called components. Only half of the mappings have sheets, the others are implemented by the XPrez library which is embedded in Proxima.

Reviewing diagram 2, it should immediately be clear that quite a few layers are hidden from the user. A user would only write the user components and 'plug' them into Proxima. All components except the data declarations can be written in Haskell, but the presenter component is usually written in AG.

The relation depicted by the lolly pop arrow is analogous to Haskell function applications. Consider for example the editor's main function which directly calls Proxima with the user components as arguments, e.g.

```
main :: IO ()
main = proxima evaluationSheet reductionSheet presentationSheet
               parseSheet scannerSheet initDoc initEnr
```

The components would, in this case, be represented by functions, not Haskell modules.

The 'uses' relation can be understood as a Haskell import statement. The diagram shows that not only is the generated code used in the user components, Proxima itself also depends on it. This dependency unfortunately prevents it from being compiled as a separate Haskell library, but that will change in future versions of Proxima.

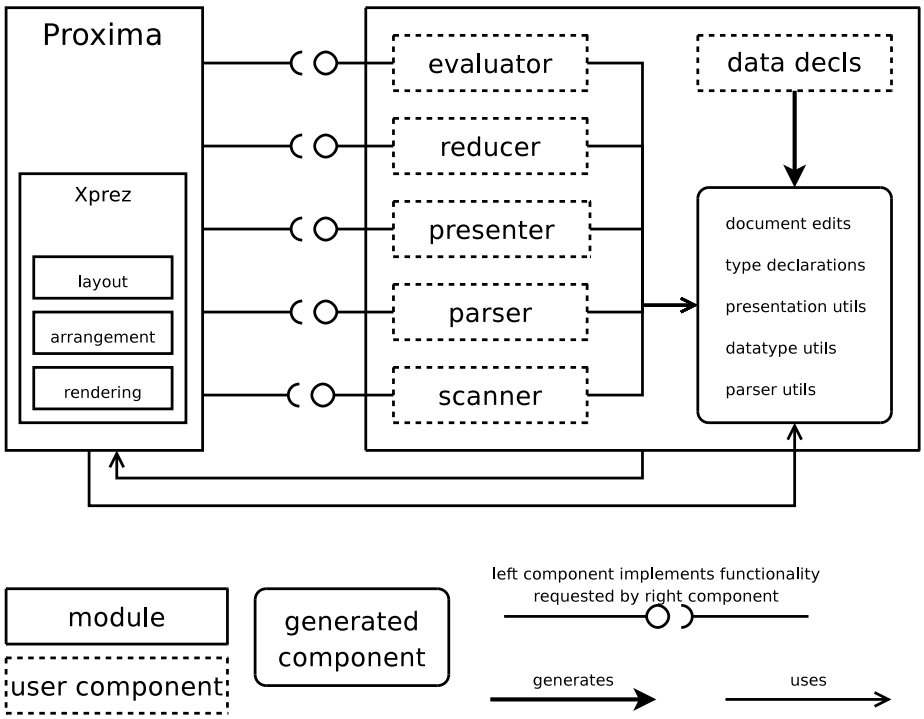

Figure 2: A component oriented view of Proxima

3 Casus description

The aim is to create a stand-alone editor starting from a very basic slide-show editor and extend it, step-by-step, with additional features. The separate features should allow us to subdivide our case study further, where ideally each feature addition will test a unique aspect of Proxima. Some features however will (and should) have cross-cutting implementations, i.e. utilize multiple aspects of Proxima.

3.1 A slide-show

Apart from the editor, the slide-shows themselves have to be very limited as well. This paper concentrates on the textual and structural edit operations rather than the visual, presentation-oriented operations. Therefore no information on the presentation, like font face, size, or color, is defined in the document. Only the list type (alphanumeric, numeric, or bullets) can be considered a presentation related aspect.

A systematic description of a slide document would read as follows:

- the slide-show's title
- the author
- a list of slides, each of which contains
  - a slide title
  - a list of paragraphs and item-lists (one mixed list of paragraphs and item-lists), each of which contain
    - in case of a paragraph, just text
    - in case of an item-list, the list type and again a list of paragraphs and item-lists

3.2 The editor's requirements

The basis of the editor should be as simple as possible, even loading and saving of files is considered a separate feature. With this, the start-up costs of a Proxima project can be determined and boilerplate code can be identified. The following features shall be implemented:

- Loading and saving files
- Adding and removing slides
- Generated title and overview slides
- A full-screen slide view

4 A bare boned editor

As stated in the requirements, the minimal editor should provide us with a clean slate, rather than extend the existing editor. It would probably have been best to start from scratch, unfortunately Proxima's complexity makes that unfeasible at this point. Therefore the slide-show editor is based on a stripped down version of the original editor prototype.

4.1 The `data declarations`

The data declarations are easily defined. Just translating the document structure presented in the requirements into the Haskell like data declarations will do. However, every data declaration's alternative must also have an `idd` field added because the generator depends on it.

```
data EnrichedDoc = RootEnr slides : [Slide]
                           fullscreen : Bool
                           currentSlide : Int
                           title : [String]
                           author : [String]
                           { id : IDD }

data Slide = Slide title : [String]
                   body : [Paragraph]
                   { idd : IDD }

data Paragraph = Text     words : [String]
                          { idd : IDD }
               | ItemList listType : ListType
                          items : [Paragraph]
                          { idd : IDD }

data ListType = Bullet { idd : IDD }
              | Number { idd : IDD }
              | Alpha  { idd : IDD }
```

The normal document, defined in pure Haskell, practically is the same as the enriched document. Only the root nodes are distinct. This will simplify the evaluator and reducer considerably.

4.2 The `evaluator` and `reducer`

The evaluator should be very simple. Only the root nodes have to be mapped from and to the enriched document's root node. The following lines show how simple this operation really is:

```
docLvlToEnrLvl (DocumentLevel rootDoc focus _) =
```

```
  EnrichedDocLevel (rootDocToEnr rootDoc) focus

rootDocToEnr (RootDoc idd slidelist fullscreen currentSlide title author) =
  RootEnr idd slidelist fullscreen currentSlide title author
```

Unfortunately some of the standard editing functions implemented by Proxima depend on the evaluator to apply the functions. For example moving the cursor with the arrow keys is handled partially by the evaluator. The code performing these actions can be copied verbatim from the prototype. Ideally this code would be moved to the framework in future versions of Proxima.

Although both the evaluator and reducer are conceptually very simple, but their types are quite daunting, e.g. the reducer has type

```
reductionSheet ::
  LayerStateEval
  -> EnrichedDocLevel EnrichedDoc
  -> DocumentLevel Document clip
  -> EditEnrichedDoc documentLevel EnrichedDoc
  -> IO ( EditDocument documentLevel Document
        , LayerStateEval
        , EnrichedDocLevel EnrichedDoc )
```

While the simple slide-show editor only needs the second argument and the first element of the function's result, more complicated editors might need this complicated interface. Note that Proxima's standard editing functions also depend on some of the other arguments.

4.3 The `presenter`

Once the interaction between the presentation and parser is understood, the presentation is quite simple. One should keep in mind that the whole document has to be derived from the presentation, even 'invisible' properties like the list type. To this end so called structural nodes are used in combination with locaters. The locater `loc` contains the slide's AST, allowing a full reconstruction of the slide. A `structural` combinator marks the slide as a structural node, rather than a parseable textual node. In this case the exact document structure was translated to structural/locater nodes. For example a slide is represented by:

```
loc (SlideNode @self @lhs.path)
$ structural
$ presentFocus @lhs.focusD @lhs.path
$ presentSlide @title.graphicPres @body.graphicPres
```

where `presentFocus` and `presentSlide` are functions dealing with the presentation of focus and the actual slide respectively.

4.4 The `scanner`

Before the presentation can be parsed, the presentation tree has to be flattened where possible. For example rows and columns will be flattened in a top-left to bottom-right fashion. All remaining nodes will be structural nodes, parsing nodes and text nodes. As the user's input is split up into individual characters, it is the scanner's responsibility to reconstruct the original strings.

The Haskell editor comes with a fully tokenizing scanner. Although this tokenizer has a quite generic set-up, it cannot be used for the slide-show editor because it loses important white space information. Furthermore, the slide-show's presentation will only have simple strings to parse. A full tokenizer is not needed. Instead a very simple scanner that only simplifies the presentation tree suffices. This scanner leaves only parsing, structural, and reconstructed text tokens.

Because of type restrictions, the actual scanning mechanism is included in the parser module. This is a known issue that happens to stand out particularly in the slide editor; the scanner of the Haskell editor prototype is mostly implemented in the scanner module.

4.5 The `parser`

The parser is defined using a parser combinator library. To allow the parser to operate on trees, Proxima adds primitives for structural and parsing nodes. Only text tokens in the parsing nodes are parsed. Structural node tokens contain a copy of the original enriched document, which can be retrieved through generated `reuse` functions. This way, even if only part of a document is displayed as parseable text, it can still be recovered by the parser.

For example the slide recognizer (structure parsers are called recognizers) is defined as follows:

```
recognizeSlide :: ListParser Document Node ClipDoc Slide
recognizeSlide = pStr $ reuse
      <$> pStructural SlideNode    -- parse structural node
      <*> recognizeList_String_    -- followed by a title
      <*> recognizeList_Paragraph  -- followed by the contents
  where reuse str title itemList =
          reuseSlide [tokenNode str]
                     Nothing
                     (Just title)
                     (Just itemList)
```

where `reuseSlide` is a generated function constructing a slide with optional arguments. The omitted (with value `Nothing`) arguments will be filled in with the original values. Note the two structural node parsers `pStr` and `pStructural`. The former is used as a wrapper for the parsers of the structural's children, while the latter is used to capture the hidden enriched document to be reused.

4.6 Conclusions

Quite a bit of code is required to get started. Part of this is inevitable, for example the presentation is unique for this application, but some code should really be defined in the framework. Especially the evaluator and reducer contain a lot of code that implements functionality that only concerns Proxima.

There is some code that should be very similar for many editors, yet small differences prevent it from being put into the framework. For example the structure parser has some patterns that will be very similar for many editors. Some of this code might be generated by using a more flexible generator (see 6.1.

5 Adding features

This section describes the experience of adding features to the editor in Proxima. This follows a fixed pattern. For every feature

1. the design choices regarding the feature are discussed first
2. then the changes are listed for every component, omitting only the smaller details
3. finally the weaknesses and strengths of Proxima with respect to the feature will be discussed next, in some cases combined with suggestions for improvement

5.1 Loading and saving files

To enable saving and loading of slide-shows, a file format has to be chosen. Because Proxima has special support for XML, it seems best to use XML as the editor's target language.

5.1.1 Implementation

Proxima's generator always generates an XML document parser and pretty printer. Thanks to this, the implementation takes a mere seven lines of code. The code just calls the generated functions with the proper arguments and, in case of the parser, handles possible parse errors.

5.1.2 Conclusions

The generator really shows its usefulness here. The only downside is the lack of a readable interface. The generated code has to be inspected to find out what functions need to be called and how they should be called.

5.2 Adding and removing slides

Proxima has special support for context menu items (right click menu). Inserting slides clearly is a context sensitive operation, it seems therefore a natural solution. Thus clicking right on a slide shall give two options:

1. remove this slide
2. insert a slide after the current slide

5.2.1 Implementation

Context menu items can be defined in the presentation with a special combinator `addPopupItems`, which adds a list of operations to the context menu of a presentation element. These operations are defined as a transformation of the enriched document.

```
slidePres `addPopupItems`
   [ ("Insert new slide" , slideAdder @lhs.ix)
   , ("Remove this slide", slideRemover @lhs.ix)]
```

where `slidePres` is the slide's presentation, `slideAdder` and `slideRemover` are the transformation functions and `@lhs.ix` is the current slide's number.

5.2.2 Conclusions

One is in no way concerned with the details of the context menu's inner workings. The implementation couldn't be more straightforward.

5.3 The title and contents slides

The title page is quite straightforward; there are no real decisions to make. An editable contents page however is more interesting, since it is a secondary view on the slides themselves. For example, can slides be removed, reordered, or added in the contents page as well? To keep things simple, the choice has been made to only allow textual changes to the slides' titles.

5.3.1 Implementation

The title slide is implemented by a simple extension of the presentation and the parser. A contents slide however, is more complicated. The presentation is extended with a contents slide, in which every slide title is presented by its title and its structural node.

The parser is extended to parse the new slide and compare the parsed titles with the original version found in the structural nodes. If the titles in the contents slides have changed, the changes will be applied to the parser's result (an enriched document).

5.3.2 Conclusions

Comparing the original document with the parsed result to detect changes doesn't seem to belong in the parser, but there is no natural place to do this. Generic detection of changes will fortunately be added to a future version of Proxima.

5.4 A full-screen view

The full-screen view of the slides captures the final purpose of the slide editor: producing slides to be projected during a slide show. While one might argue that the slides should not be edited during a presentation, it is more interesting to test Proxima with a fully editable full-screen view.

Unfortunately XPrez does not have any control over the window containing the editor and no support for scaling. For this reason the full-screen view is implemented as a single slide view instead.

As for the interface, adding an item to the window menu in the 'View' sub-menu should suffice. To proceed to the next slide, the page down button can be utilized and page up for the reverse.

5.4.1 Implementation

Two new items are added to the document types (both enriched and original):

1. A boolean that is set to true if and only if the full-screen view is enabled.
2. The slide number that is currently being viewed. This only has any meaning when full-screen view is enabled.

Defining the presentation of the full-screen view is quite simple, but the interaction with the parser is not. All slides are included in the presented document in hidden structural nodes. The parser uses these to reconstruct the complete slide-show from the presentation of only a single slide.

Adding an item to the window menu and enable the keyboard interaction can only be achieved by editing Proxima. Not only in the GUI definition, but each layer's types have to be adjusted to accommodate the additional information. For example the layout layer was extended as follows:

```
data EditLayout documentLevel doc node clip =
    SkipLay Int
  | SetFocusLay FocusPres
  ...
  | ToggleFullscreenLay
  | PageUpLay
  | PageDownLay
```

All of Proxima's mappings have to be changed, for example the mapping of the layout onto the presentation need to translate `PageDownLay` to `PageDownPres`. The evaluation function finally adjusts the document's values.

5.4.2 Conclusions

Adding two, view-related items to the document types affect the generated code in ways that might not be intended. The generated XML view will include the two new values and so will the load and save functions. Since this approach is also taken in the prototype, it seems the intended solution for situations like these. It would be preferable to hide layer-to-layer communication from other layers, rather than extending the document's data type.

Obviously the current situation, where Proxima has to be extended to handle keyboard input, has to be dealt with. The framework could parameterize all user input, allowing an editor writer to process it directly. There seem to be two main approaches:

1. Extend all layers to include input signals, which would be passed upwards from the rendering layer to the document layer. For example by extending the current `Edit` data structures with a general input data element:

   ```
   data EditLayout documentLevel doc node clip input =
       SkipLay Int
     | SetFocusLay FocusPres
     ...
     | UserInputLay input
   ```

   On the one hand, the implementation would be very simple and compatible with the existing work. On the other hand, this approach would lose the `Edit` data structures' current abstractions; now, the data structure contains actions with known semantics, while user input can be interpreted in any way.

2. Handle keyboard input through callbacks: The editor writer defines a number of functions that will be called by Proxima for specific key presses. A solution that could look very similar to the existing context menu handling described in 5.2.1. With type classes the same could be achieved.

Technically, it would even be possible to provide multiple solutions and let the user choose what is appropriate.

6 A review

Some of Proxima's aspects are not particular to one feature. Therefore this section reviews those aspects of Proxima that were not discussed in detail before. As well as pointing out problems, possible solutions are presented and discussed.

6.1 The generator

The generator has shown to work very well for some examples. In fact, it makes one eager for more generated code. For example quite some code in the parser

is very repetitive in nature and seems, at least partially, well suited for code generation.

There are however two main problems:

1. There is no clear interface for the generated code. Usually generated code, for example a parser generator, implements just a single function. Proxima's generator however produces a large number of functions that can be used at many places. To find out what exactly gets generated and how it should be used, one has to inspect the generated code itself.

2. Code generation is very inflexible; if a generated function is *almost* exactly what is needed, it is useless. For example the XML representation of the slide document type has no support for attributes: the author and title of a slide-show are included as child elements of the root document. The only alternative is to rewrite the entire parser, although just a small change is needed.

Interestingly, the generator is based on the well studied principle of data type generic programming. When Proxima was written, the GH (Generic Haskell) system[2] was too immature to be used in practice. It might be time to reconsider and at least have a new look at GH, because it seems a perfect solution for this problem. Its ability to work with local redefinitions could provide the needed flexibility and the design of GH would provide a clear interface for free.

6.2 XPrez

Xprez, the part of Proxima that handles the presentation, has quite a few bugs, making it difficult to work with. The offered functionality is also somewhat limited. For example there is no control over the flow of text, i.e. where the arrow keys will go when inserting text. It follows a fixed top-left to bottom-right pattern.

Before XPrez can be extended it needs to be stabilized and debugged. That can be achieved by setting up a thorough, preferably automated, testbed. While this might take some work, it will help to ensure quality of the library. Systematic tests would also improve portability of the code. Finally, writing the tests present a good opportunity to document the exact semantics of XPrez' primitives.

6.3 The presentation parser

The most problematic code in the editor is, without doubt, the presentation parser. The use of parser combinators to parse trees is unconventional and sometimes confusing. On top of that the interaction between presentation and parsing, which is crucial to the editor, can be complicated, since they are separated by a number of partly hidden layers. At the same time there is very little room for error. Any mistake in the parser, resulting in a parse error, will crash the editor.

Finally the parser has some limitations. The fact that Proxima can only accept parsers whose result types are in specific classes, severely limits flexibility. Since the class instantiations for all the enriched document's types are generated this is not always a problem, but as soon as more complicated parsers are needed it causes problems. Therefore, removing the class requirements should be seriously considered. Perhaps different versions of the primitives `pStr` and `pPrs`, that do not introduce the class requirements, can be added.

6.4 Model, view and controller

The MVC (model-view-controller) architectural pattern dictates a very strict separation of concerns in GUI development. Its three components, can be defined as follows:

model An abstract representation of the data. This clearly corresponds with Proxima's document layer.

view A rendering of the model to be used for interaction with a user, commonly implemented by a GUI. In Proxima the view can be found in the presentation, layout, arrangement and rendering layers. Although other layers can be involved as well, as the slide-editor's full screen presentation demonstrates.

controller The collection of actions that can be performed on the model. These actions are defined in many of Proxima's mappings, but mostly in the reducer, evaluator, presenter and scanner.

While the MVC (model-view-controller) paradigm is not the only game in town, it is very well established in the GUI programming world. Proxima might benefit from its success by incorporating it.

Rather than imposing MVC, an editor programmer could be given the choice to adhere to it or not. In the current implementation, the view and controller tend to get mixed up in the presenter component (the mapping from enriched document to presentation). For example the removal and insertion of slides is completely implemented in the presenter.

To untangle the view and presentation, no actual changes to Proxima would be required. The controller's functionality, except for the presentation parser, could be inserted into the enriched document as alternative nodes. For instance, the function removing a slide could be inserted into the enriched document's data structure as depicted in figure 6.4. The left branch of the *RemoveSlide* node is the original slide list, the right branch has its head element removed. Of course, this is only a suggestion which needs thorough investigation.

7 Conclusion

During this case study, Proxima has indeed shown itself a very flexible and powerful editor framework. Even if applied to a concept very different from

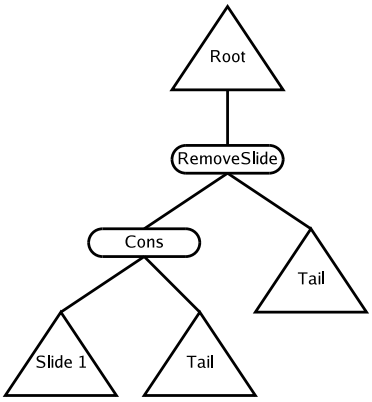

Figure 3: A flow oriented view of Proxima

traditional text-based editors, it holds its own. This strength seems to stem from the underlying architecture.

Whether this document can serve as documentation for other programmers remains to be seen. Notwithstanding the unproven value as documentation, the case study as pointed out a number of specific problems with the current implementation of Proxima. Some of these have already been addressed in recent updates of the framework, others might have to be addressed in the future.

References

[1] A. Baars, D. Swierstra, and A. Loh. UU AG System User Manual. *Department of Computer Science, Utrecht University, September*, 2003.

[2] R. Hinze and J. Jeuring. Generic haskell: Practice and theory. *Generic Programming: Advanced Lectures*, 2003.

[3] M. Schrage. *Proxima: a presentation-oriented editor for structured documents*. Utrecht University, 2004.

[4] M. Schrage and J. Jeuring. Xprez: A declarative presentation language for xml, 2003.