CHAPTER 2

Architecture of the Proxima Editor

**** Version: 30th May 2003 ****

A generic editor is a large system with an architecture that consists of many components for performing many different tasks. However, in this chapter, we focus on those components of the architecture that are involved in the process of presenting the internal document to the user, and translating the edit gestures given by the user to edit operations on the document structure. Of course, other functionality, such as IO handling, macro processing, or a search facility is very important for the usability of the final editor, but its implementation is mainly straightforward and merely requires a substantial amount of engineering. *** The presentation and handling of edit gestures, on the other hand, is of greater importance, because it determines for which applications the editor can be used, and how powerful the editing behavior will be.

*** maybe a bit to strong?

The core components of the Proxima architecture are a number of layers that only communicate with their direct neighbors. The layered structure is based on the staged nature of the presentation process. Instead of mapping it directly onto its rendering, a document is first mapped onto an intermediate data structure. The intermediate data structure is mapped onto another intermediate data structure, until a final intermediate data structure is mapped onto the rendering. The intermediate data structures are the data *levels*, and the components that take care of the mapping are the *layers*. Figure 2.1 schematically shows the levels and layers of Proxima. Only two levels are visible at each layer: an upper and a lower level.

There are a number of reasons why the Proxima architecture is layered:

Staged presentation process. The presentation process is naturally staged. The process

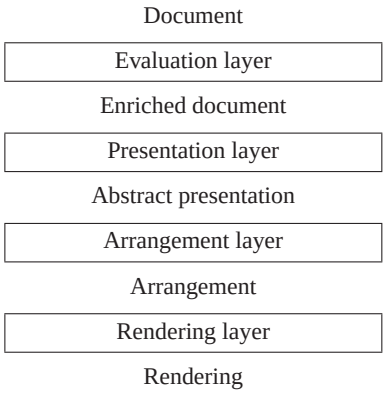

Figure 2.1: The levels and layers of Proxima (draft)

consists of repeatedly mapping structures that have a different meaning on a higher level onto the same set of lower level structures. For example, a table of contents, once its structure has been computed, can be presented in the same way as a chapter structure. Similarly, a line that comes from a formatted paragraph is rendered in the same way as a line that was explicitly specified in the presentation. Mappings like these form stages in the presentation process that can be performed by separate layers.

Specification of presentation and edit behavior. A layered architecture provides natural hooks for the editor designer to specify presentation and edit behavior. A separate evaluation layer, for example, makes it possible to separate computation and presentation, thus allowing different style sheets to be used for a document together with its derived structures. At the same time, layers offer more control over backward mappings, for example specifications of how edit operations in derived structures should be translated to edit operations on the document.

∗ ∗ ∗ maybe this name is not right: "Extra state"?

Local state. *** An important aspect of the Proxima editor is the concept of local state, which is inherently connected to a layered architecture. Each level in the presentation process may contain information that is not present in the surrounding upper or lower levels. Applications of local state include keeping track of the focus and storing whitespace for tokens.

Keeping bidirectional mappings. Because Proxima supports editing on all levels, a mapping between each pair of levels needs to be maintained. Maintaining such mappings is easier in a layered architecture. Furthermore, the lower layers can maintain the mappings automatically.

Efficiency. Some steps in the presentation process, especially in the higher layers, may be time consuming because global computations need to be performed. In a layered

architecture, it is possible to perform the higher layer computations not at every keystroke, but only once in a while. For example, in a program editor, parsing the program may be delayed until the user enters a whitespace character or performs a navigation operation. Type checking the program may be delayed until after a certain period of inactivity, or when specifically requested by the user.

The remainder of this chapter contains an informal description of the levels and layers in the Proxima architecture. A formal description is presented in Chapter **??**.

2.1 The levels of Proxima

A data level in Proxima is not just an intermediate value in the presentation computation, but an entity in its own right. Together, the data levels constitute the state of the editor. The six data levels of Proxima are:

Document: The edited document, the type of which is described by a DTD or an EBNF grammar.

Enriched Document: The document enriched with computed information.

Presentation: A logical description of the presentation of the document, consisting of rows and columns of presentation elements with attributes. The presentation also supports elements to be specified to be formatted based on the available space (line breaking).

Layout: Similar to the presentation, but with explicit whitespace.

Arrangement: Similar to the layout, but with absolute size and position information. At this level, line breaking have been performed.

Rendering: A collection of user interface commmands for drawing the absolutely positioned and sized arrangement.

2.1.1 Document

A document is the internal tree data structure that is edited by the user. The type of the document is described by a context free grammar, with special constructs for lists and optional values (similar to EBNF). Haskell data types, EBNF, DTDs and XML Schemas are all restricted forms of context free grammars suitable for describing the document type. In this thesis, we make use of simple Haskell data types without higher-order types, except for the list and the maybe type.

The exact type formalism is important for document to document transformations (ie. document edit operations), because it should be possible to guarantee type safety of such

transformations. However, for the time being, the only supported document edit operations are simple tree based operations, such as cut and paste, and basic operations on lists and optionals, such as selecting a segment of a list. Therefore, using a context free grammar formalism for describing document structure is powerful enough.

Because the type of the document will vary for different applications of the editor, we can only give an example for a specific application. The example document consists of a list of declarations, each of which is an identifier declaration or a comment. An identifier declaration contains a string that represents the declared identifier, as well as an expression that may contain conditional expressions, integers, and booleans. The third field of the declaration is a string that may contain additional information about the declaration. It is used to illustrate the local state concept in the Proxima editor. A comment consists of a list of strings. The types `String`, `Int`, and `Bool` are primitive types. Although not very suitable for practical purposes, the chosen document type will allow us to illustrate the different aspects of the Proxima data levels and layers.

```
data Document = Root_Doc [Decl_Doc]
data Decl_Doc = Decl_Doc String Exp_Doc String
              | Comment_Doc [String]
data Exp_Doc = IfExp _Doc Exp_Doc Exp_Doc Exp_Doc
             | IntExp_Doc Int
             | BoolExp_Doc Bool
```

An example document of this type, as well as examples of the lower levels, is provided with the explanation of the presentation process in Section 2.3.

2.1.2 Enriched Document

An enriched document is a copy of the document, to which derived information has been added. In the word processor example, the enriched document contains a table of contents, and each section or subsection element has a field that contains its number. Such derived information is not present in the document level.

Besides containing extra information, the enriched document, or a subtree of it, may also be a reordered version of the document. For example, the enriched document may contain a sorted list of values, which is not sorted in the document.

As an example of an enriched document, we take the sample document of the previous section and a type declaration alternative to the *Decl* type. The type declaration is computed for each declaration. The type of an expression may be integer, boolean, or erroneous (eg. `if True then 0 else False`). It is also possible to add the type as a field to the *Decl* alternative, however, separate type declarations make it easier to show how edit operations on the enriched document are handled in Section 2.4.5.

```
data EnrichedDocument = Root_Enr [Decl_Enr]
```

```
data Decl_Enr = TypeDecl_Enr String Type_Enr
              | Decl_Enr String Exp_Enr String
              | Comment_Enr [String]
data Exp_Enr = IfExp _Enr Exp_Enr Exp_Enr Exp_Enr
             | IntExp_Enr Int
             | BoolExp_Enr Bool

data Type_Enr = IntType_Enr | BoolType_Enr | ErrorType_Enr
```

2.1.3 Presentation

A presentation is an abstract description of what the document will look like to the user. It consists of strings, images, and simple graphical elements (lines, rectangles, etc.) that are grouped in rows, columns and matrices. Elements have attributes for colors, line styles, fonts, and alignment. Attribution can be influenced using a *with* element, which contains a rule that specifies how the attribution is affected.

There are three ways of positioning elements in the presentation. Firstly, the position can be specified relative to other elements in the presentation, by placing a list of elements next to each other in a *row*, or above each other in a *column*. Elements are aligned according to reference lines (eg. the baseline for a string), and stretchable elements may be used to influence the positioning. Besides rows and columns, a *matrix* construct presents a list of lists of elements aligned both horizontally as well as vertically, and an *overlay* presents a list of elements in front of each other (eg. for presenting a squiggly line).

The second way of positioning presentation elements is by using a *formatter* element, that positions a list of children based on the available space. Currently, Proxima only supports horizontal formatting, suitable for line breaking in a paragraph. Vertical formatting is not fundamentaly different, but has not been implemented yet. Furthermore, support for a page model also requires extensions to the lower levels, which have not been realized yet.

Finally, a presentation can consist of a list of tokens, which may be identifiers, operators, integers, strings, etc. If textual editing on a presentation is desired (eg. for a source editor), a parser is invoked on the presentation after it has been edited. A presentation that needs to be parsed may be specified using tokens, which make it possible to use a separate layer for scanning. A token contains information about the whitespace (line breaks and spaces) preceding it in the presentation. In some cases, a presentation may need to be parsed without using the Proxima scanner. For example, when explicit whitespace information is stored in the document. In that case, dummy tokens that are not processed by the scanner may be used for the presentation.

In some cases, a presentation that needs to be parsed contains parts that we do not want to be parsed. For example, non-textual presentations, such as images. Such presentations can be included in the token list presentation with a *structural token*. A structural token contains a presentation, and is treated specially by the parser. Although the child presentation is not parsed, it may be an arbitrary presentation, and hence contain a token list

presentation itself, that will be parsed.

Unlike the document and the enriched document, the presentation has a fixed type. It is discussed in more detail in Chapter **??**. Here, we present a slightly simplified subset of the type. The details regarding the attribution (eg. color, font, and reference lines) of presentation elements have been left out, by leaving the type `AttributionRule` abstract.

```
data Presentation = Empty_Pres
                  | String_Pres String
                  | Tokens_Pres [Token]
                  | Row_Pres [Presentation]
                  | Column_Pres [Presentation]
                  | Overlay_Pres [Presentation]
                  | Matrix_Pres [Presentation]
                  | Formatter_Pres [Presentation]
                  | With_Pres AttributionRule Presentation

data Token = UCaseToken Whitespace String
           | LCaseToken Whitespace String
           | IdentToken Whitespace String
           | OpToken Whitespace String
           | IntToken Whitespace Int
           | StructuralToken Whitespace Presentation
           ...

type Whitespace = (LineBreaks, Spaces)
type LineBreaks = Int
type Spaces = Int

data AttributionRule = ...
```

*** *** *** ***

* * * refnr in row/col?

* * * Hoe doen we tokens?

* * * verschillende types Token vs Presentation?

* * * mention Structural tokens?

2.1.4 Layout

The layout level is the same as the presentation level, but without the tokens. Whitespace information from the tokens is represented in the layout level by strings containing spaces and by starting a new row for each line break. Formatters are still present, because the exact size and position information required to remove them is not known at the layout level.

The similarity between the layout and the presentation level is clearly visible in the types: the `Layout` type is the `Presentation` type without the `Tokens` alternative.

```
data Layout = Empty_Lay
```

```
            | String_Lay String
            | Row_Lay [Layout]
            | Column_Lay [Layout]
            | Overlay_Lay [Layout]
            | Matrix_Lay [Layout]
            | Formatter_Lay [Layout]
            | With_Lay AttributionRule Layout

data AttributionRule = ...
```

2.1.5 Arrangement

At the arrangement level, each element gets its position and size. The position is expressed in actual coordinates. These may be different from the pixel coordinates in the final rendering because the rendering may be scaled. Formatters have been resolved at this point, and are represented by columns of rows.

The arrangement is structured as a tree and positions are relative to the parent element. Relative positioning makes it easier to reposition a subtree in the arrangement, because it removes the need to update the position of each element in the moved subtree.

A with node does not have a geometry field, because it only influences the attribution of the tree, but does not form an actual part of the arrangement. ***

*** geometry for Empty?

```
*** two options:  position in each child, or positions in parent ***

data Arrangement = Empty_Arr Geometry
                 | String_Arr Geometry String
                 | Row_Arr Geometry [Arrangement]
                 | Column_Arr Geometry [Arrangement]
                 | Overlay_Arr Geometry [Arrangement]
                 | Matrix_Arr Geometry [Arrangement]
                 | With_Arr AttributionRule Arrangement

type Geometry = (Position, Size)
data AttributionRule = ...

*** of ***

data Arrangement = Empty_Arr
                 | String_Arr Size String
                 | Row_Arr Size [Arrangement] [Position]
                 | Column_Arr Size [Arrangement] [Position]
                 | Overlay_Arr Size [Arrangement] [Position]
```

```
                    | Matrix_Arr Size [Arrangement] [Position]
                    | With_Arr AttributionRule Arrangement

type Position = (Int, Int)
type Size     = (Int, Int)
data AttributionRule = ...
```

∗ ∗ ∗ have to make a choice: 2nd one makes moving arrs in a compound arr easier, good for incrementality, but has not been implemented/tested

∗∗∗

2.1.6 Rendering

A rendering is a set of user interface drawing commands that are actually drawn on the screen. Positions are expressed in pixel coordinates. In contrast to the other levels, a rendering has no explicit tree structure. ∗∗∗

∗ ∗ ∗ maybe it should be a tree though

Because the rendering is highly dependent on the GUI library that is used, we only give an abstract type. ∗∗∗

∗ ∗ ∗ Haskell syntax? (type vs. data)

```
type Rendering = [RenderingCommand]
data RenderingCommand = ...
```

2.2 The layers of Proxima

Between each pair of data levels is a *layer* that takes care of mapping the upper level onto the lower level (downward mapping, or presentation) and that translates edit operations on the lower level to edit operations on the upper level (upward mapping, or translation).

In the actual architecture, the downward mapping is not a mapping between the upper and lower levels, but between edit operations on the levels, similar to the upward mapping. Hence, the upward and the downward mappings are symmetrical, as both are concerned with edit operations. The layer has access to the upper and lower levels surrounding it, and may update the upper level. The exact types of the mappings in a layer are presented in Chapter **??**. Figure 2.2 contains a more detailed picture of a single layer. The update on the higher level is denoted with the $\rightsquigarrow$ symbol.

∗ ∗ ∗ Mention where update takes place?

∗∗∗

A layer has two separate components, one for the downward mapping (presentation) and one for the upward mapping (translation). Because each layer is between two levels and the Proxima system consists of six data levels, there are five layers:

- Evaluation layer: Evaluator/Reducer
- Presentation layer: Presenter/Parser

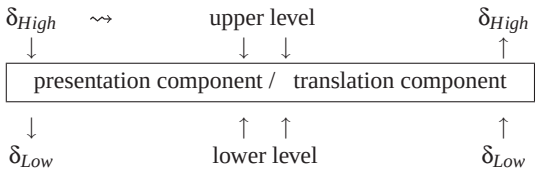

Figure 2.2: A single proxima layer

- Layout layer: Layouter/Scanner
- Arrangement layer: Arranger/Unarranger
- Rendering layer: Renderer/Gesture Interpreter

The downward mappings together form a logical whole (the presentation process), just like the upward mappings (the translation process). Therefore, rather than discussing the components pairwise from layer to layer, we give a description of all the components involved in the presentation process, followed by the components of the translation process.

2.3 Presentation Process

The presentation process is the stepwise mapping of a document onto its final rendering. Although the mapping is actually a mapping between edit operations on each of the levels, we will present it here as a mapping between edit levels. The reason for this is that the fact that the mapping is on edit operations rather than on levels is important mainly for reasons of incrementality, and regarding it as a mapping on levels makes the presentation process easier to explain.

In order to illustrate the different stages in the presentation process, we follow the presentation of an simple document. After the description of each layer component, the intermediate result of the presentation of the document is given.

The document type of the example is the expression list document type from Section 2.1.1. The sample document consists of two items: a comment and an expression. For sake of clarity, we will denote strings on each level with `"..."` instead of $\texttt{String}_{level}$ `"..."`.

Document:

```
Root_Doc [ Comment_Doc ["This", "is", "a", "simple", "expression"]
         , Decl_Doc "simple1"
                    (IfExp_Doc (BoolExp_Doc True) (IntExp_Doc 1) (IntExp_Doc 0))
                    "info"
         ]
```

2.3.1 Evaluation layer: Evaluator

The first step in the presentation process is the computation of the derived information in the document. The component that takes care of this is the evaluator. The evaluator is parameterized with a *computation sheet*, which is a declarative specification of the derived values. The computation sheet is specified using the attribute grammar formalism.

Besides basic values, such as section numbers or the outcome of a computation in a spread sheet, the evaluator may also derive tree structures, such as a table of contents. The derived structures may be partial, duplicated, or reordered versions of the document. ***

∗ ∗ ∗ probably more is possible

Example: For each declaration, the evaluator computes a type declaration in the enriched document. In the example this means that a type declaration for `"simple1"` with type `IntType` is included in the item list, yielding:

Enriched document:

```
Root_Enr [ Comment_Enr [ "This", "is", "a", "simple", "expression" ]
         , TypeDecl_Enr "simple1" IntType_Enr
         , Decl_Enr "simple1"
                    (IfExp_Enr (BoolExp_Enr True) (IntExp_Enr 1) (IntExp_Enr 0))
                    "info"
         ]
```

2.3.2 Presentation layer: Presenter

The enriched document is mapped onto the presentation by the presenter. Similar to the evaluator, the presenter is parameterized with a *presentation sheet* that specifies the presentation. The presentation sheet is an attribute grammar that defines the presentation as a synthesized attribute for each element in the enriched document.

Tokens form a special case in the presentation process, because a token contains its own whitespace, which is not represented in the document or enriched document levels. Therefore, the presenter keeps track of which tokens the enriched document is mapped onto. If a enriched document structure is re-presented, the old tokens (and corresponding whitespace) are reused.

If an enriched document element that is presented with tokens, is newly created or has not been presented before, a default value for the whitespace of its tokens is chosen. This default may come from a pretty print algorithm. A default value may be used also in case a structure has been edited in such a way that reusing the old tokens does not make sense.

The mechanism of reusing presentation tokens is also used to handle ambiguities in token representations. For example when a user has entered the text ”001” which is stored in the document as the integer 1, the mapping between the enriched document and the presentation ensures that on re-presentation, the integer is presented as ”001” instead of ”1”

The evaluator and the presenter both make use of the attribute grammar formalism and are closely related. The separation between evaluating and presenting is not strict. On the one hand, the entire presentation can be regarded as a derived structure, and on the other hand the presentation sheet can reorder elements in the presentation and introduce structures, which is more appropropriately done in the evaluation sheet.

The editor designer must make a careful decision on where to specify document evaluation and presentation. Whenever it is conceivable that a derived structure may have different presentation styles, the computation of the structure and presenting it is best separated. A good example of such a structure is a table of contents, whose presentation must match the presentation of the document. Another case in which separation is beneficial is when edit operations on the derived structure have to be supported, as the separation makes it easier to specify the translation of the edit operations. ***

* * * bad paragraph

Example: The example presentation of the enriched document is basic: a comment is put in a formatted paragraph and a (type) declaration is presented in a textual infix representation using tokens. The third string field of the declaration `"info"` is not included in the presentation in order to have an example of a partially presented structure. The reason for this becomes apparent in Section 2.4.4 on parsing. The pair of numbers in each token represents the whitespace (*nr. of line breaks*, *nr. of spaces*) preceding it.

The `With` nodes specify the font for the presentation of the declarations. For sake of simplicity, the exact details of the attribution rule are not shown: the bracketed declaration {`fontFamily = "`*name*`", fontSize =` *size*} specifies the font family and size for the child of the with node. *** ***

* * * need an end token

* * * mention that this is a simple whitespace repr. that does not allow lines containing only spaces?

Presentation:

```
Col_Pres [ With_Pres { fontFamily = "Times New Roman", fontSize = 12 }
             (Formatter_Pres [ "This", "is", "a", "simple", "expression" ])
         , With_Pres { fontFamily = "Courier New", fontSize = 12 }
             (Tokens_Pres [ LCaseToken_Pres (1,0) "simple1", OpToken_Pres (0,1) "::"
                          , UCaseToken_Pres (0,1) "Int"
                          ])
         , With_Pres { fontFamily = "Courier New, fontSize = 12 }
             (Tokens_Pres [ LCaseToken_Pres (1,0) "simple1", OpToken_Pres (0,1) "="
                          , LCaseToken_Pres (1,2) "if", UCaseToken_Pres (0,1) "True"
                          , LCaseToken_Pres (0,1) "then", IntToken_Pres (0,1) "1"
                          , LowercaseToken_Pres (1,10) "else", IntToken_Pres (0,1) "0"
                          ])
         ]
```

2.3.3 Layout layer: Layouter (better name?)

The layouter removes the tokens in the presentation level, yielding the layout level. Each list of tokens is mapped on a column that contains rows of strings. Each token is represented as a string *** and spaces are represented as strings of spaces. A line break causes

* * * Structural tokens are not strings

∗ ∗ ∗ mention a layout sheet?

a new row in the layout. ∗∗∗

Example: The tokens that are present in the presentation level are replaced by strings in the layout level (the ␣ character denotes a space). The formatter is not affected. The line break before the type declaration is represented by an empty string.

Layout:

```
Col_Lay [ With_Lay { fontFamily = "Times New Roman", fontSize = 12 }
            Formatter_Lay [ "This", "is", "a", "simple", "expression" ]
        , With_Lay { fontFamily = "Courier New, fontSize = 12 }
            (Col_Lay [ ""
                     , Row_Lay [ "simple1", "␣", "::", "␣", "Int" ]
                     ])
        , With_Lay { fontFamily = "Courier New, fontSize = 12 }
            (Col_Lay [ Row_Lay [ "simple1", "␣", "=" ]
                     , Row_Lay [ "␣␣", "if", "␣", "True", "␣", "then", "␣", "1" ]
                     , Row_Lay [ "␣␣␣␣␣␣␣␣␣", "else", "␣", "0" ]
                     ])
        ]
```

2.3.4 Arrangement layer: Arranger

The arranger computes the exact sizes and positions for all elements in the layout level, yielding the arrangement. Fonts are queried to determine the size of strings, and child elements of compound elements such as rows and columns are aligned and positioned.

The arrangement layer also processes formatters by mapping them onto columns of rows in the arrangement. First, the amount of available space for a formatter is computed, and then the child elements are distributed along rows using a (possibly optimal) line breaking algorithm.

Example: The formatter, which is the first element in the top-level column of the example layout, is replaced by a column of rows in the arrangement. Furthermore, each element in the arrangement tree contains has an exact size and a position relative to its parent, which are denoted with superscripts: $element^{(x,y)(width \times height)}$.

Note that the with elements are not removed, because the font information is required to render the arrangement.

Arrangement:

```
Col_Arr^(0,0)(80×84)
   [ With_Arr { fontFamily = "Times New Roman", fontSize = 12 }
       (Col_Arr^(0,0)(80×24)
          [ Row_Arr^(0,0)(80×12) [ "This"^(0,0)(17×12), "is"^(25,0)(6×12), "a"^(41,0)(4×12)
```

```
                        , "simple"(53,0)(27×12)]
        , Row_Arr(0,12)(80×12) [ "expression"(0,0)(42×12)]
        ])
, With_Arr { fontFamily = "Courier New, fontSize = 12 }
    (Col_Arr(0,24)(75×24)
        [ ""(0,0)(0×12)
        , Row_Arr(0,12)(75×12) [ "simple1"(0,0)(35×12), "␣"(35,0)(5×12), "::"(40,0)(10×12)
                               , "␣"(50,0)(5×12), "Int"(55,0)(15×12) ]
        ])
, With_Arr { fontFamily = "Courier New, fontSize = 12 }
    (Col_Arr(0,24)(80×36)
        [ Row_Arr(0,24)(50×12) [ "simple1"(0,0)(35×12), "␣"(35,0)(5×12), "="(40,0)(5×12) ]
        , Row_Arr(0,36)(80×12) [ ... ]
        , Row_Arr(0,48)(80×12) [ ... ]
        ])
]
```

2.3.5 Renderering layer: Renderer

The renderer maps each element of the arrangement onto a set of drawing commands for the user interface. All positions and size have already been computed by the arranger, and the renderer only scales these positions and sizes according to the current scaling factor of the view.

Example: The result of applying the renderer to the example arrangement is a set of rendering commands which display the comment and the declaration when executed. Note that the comment is rendered in a different font than the declaration. The fonts are specified in the presentation, layout, and arrangement levels, but have been

Rendering:

This is a simple expression

```
simple1 :: Int
simple1 =
  if True then 1
          else 0
```

2.4 Translation Process

The translation process of edit operations is layered in the same way as the presentation process. However, there are important differences between the two.

For the translation process, edit operations are the main focus, and therefore we do not regard a translation mapping as a mapping between levels, like we did for the presentation process.

Another difference is that in the translation process, layers may be skipped. For example, a document edit operation is passed on by the lower layers, until it reaches the evaluation layer, where it is performed on the document level.

The translation of edit operations can take place either directly, or indirectly by applying the edit operation to a lower level, using a mapping between levels to compute the corresponding upper level and computing the upper level edit operation by taking the difference between the new and the previous upper level.

An example of the direct translation is when a mouse click on an absolute position in the arrangement is mapped onto a mouse click operation on a tree path in the layout level. An example of an indirect translation is the insertion of a character in the presentation level. Rather than mapping this edit operation immediately on an enriched document edit operation, the character is inserted in the presentation, the presentation is parsed, and the edit operation is distilled from the newly parsed enriched document.

On the lowest two layers (rendering and arrangement), only direct translation takes place, whereas on the higher levels also indirect translation is possible. The reason for this is that the rendering and the arrangement level cannot be edited by the user, and an indirect translation only takes place when a lower level is edited and the edited level is then mapped onto the new upper level. The lowest level that may be edited by the user is the layout level. However, the architecture does not fundamentally prohibit editing the lower levels, and a future version of Proxima may support editing on the arrangement, by allowing a user to change the absolute positions of arrangement elements. Editing the rendering seems to be rather far fetched, since the rendering is a set of GUI-specific commands.

Because the translation of edit operations does not go through all stages like the presentation does, and because of the variation in edit operations, it is not possible to give a running example of the translation process. A number of separate examples are therefore provided together with the descriptions of the translation components.

2.4.1 Renderering layer: Gesture Interpreter

The gesture interpreter has two tasks. It maps edit gestures onto edit operations for the designated levels, and it translates edit operations on the rendering onto edit operations on the arrangement, which means that absolute positions in pixel coordinates are translated descaled to arrangement level coordinates. ***

*** explain more that gesture interpreter is a bit weird because it creates wrapped upper level edit ops?

Example: We will give two example translations by the gesture interpreter: a mouse click and a key press.

The mouse edit operation is a single left click at pixel coordinates (84,57) in a rendering that has been scaled to 150%. Because of the scaling factor, the coordinates are divided by 1.5 to get the arrangement coordinates.

$\texttt{MouseClick}_{Ren}\ \texttt{Left 1 (84,57))} \mapsto \texttt{MouseClick}_{Arr}\ \texttt{(56, 38) Left 1}$

The second example is a key press of the letter 'a', which is mapped onto an insert event. However, a textual insert event is targeted at the layout level instead of the arrangement level, since the arrangement level cannot be edited textually. The insert operation is therefore not of the arrangement edit type, and needs to be wrapped. The arrangement layer unwraps the insert operation and passes it on to the layout layer.

$\texttt{KeyPress}_{Ren}\ \texttt{'a'} \mapsto \texttt{Wrap}_{Arr}\ (\texttt{Insert}_{Lay}\ \texttt{'a'})$

2.4.2 Arrangement layer: Unarranger

The main task of the unarranger is mapping locations in arrangement level edit operations on locations in layout level edit operations. A location that is specified in absolute coordinates is first converted to a location in the arrangement tree, which is specified as a tree path. Subsequently, the arrangement tree path is mapped onto a layout tree path. The arrangement tree is largely isomorphic to the layout tree, except for the formatter subtrees, as these are represented by rows and columns in the arrangement. Therefore, the mapping is mainly the identity function, except for paths in rows that originate from a formatter, which are mapped to paths in the originating formatter.

Example: An mouse left click event at position (56,38) in the example arrangement from Section 2.3.4 represents a click on the string "`Int`" in the layout (Section 2.3.3). To be precise, it is a click on the left side of the letter '`I`'. If we represent a path in the arrangement tree by list of integers and a 0 denotes the first child, then this position is represented by `[1,0,1,4,0]`. *** Hence:

*** explain more?

$\texttt{unarrange (MouseClick}_{Arr}\ \texttt{(56,38) Left 1) = MouseClick}_{Lay}\ \texttt{[1,0,1,4,0] Left 1}$

2.4.3 Layout layer: Scanner

The scanner is the first layer in which the level to level mapping is important, since the layout level may be edited by the user. Edit operations targeted at the layout level are performed on the layout level, after which the level is scanned, yielding the new presentation. A presentation edit operation is computed from the new presentation.

The scanner operates only on the subtrees in the layout layer that originate from a token

list on the presentation level, while leaving other parts of the tree unaffected. A subtree, which is a column of rows, is scanned by inspecting it row by row, and recognizing the tokens that are represented by the strings in each row. For each token, the whitespace preceding it is recorded. Whitespace in the layout level is represented either by explicit whitespace characters in the strings, or by row transitions (line breaks). *** ***

* * * structural tokens?

* * * scanner sheet? (has consequences for presentation type)

* * * note the exception for comments?

Scanning is a rather localized process, so rather than re-scanning an entire token list, only the edited part of the layout level is scanned and used to compute the appropriate presentation edit commands. ***

Example: Consider the example layout level from Section 2.3.3, and assume that the edit operation on the layout is the insertion of a space between the characters 'e' and '1' in the identifer `"simple1"` of the type declaration. Of course, inserting a space here is not allowed here, since our example declarations do not have parameters.*** A parse error will therefore occur on the presentation layer, but this has no consequences for the example on the layout layer.

* * * give the grammar somewhere?

In order to compute the edit operation on the presentation, we first apply the layout edit operation to the layout level, yielding:

```
...
Col_Lay [ ""
        , Row_Lay [ "simple 1", "␣", "::", "␣", "Int" ]
        ]
...
```

Now, the scanner is invoked on the updated parts of the layout (including the whitespace parts that precede the updated part,) which gives rise to the following list of tokens.

```
[ LCaseToken_Pres (1,0) "simple", IntToken (1,0) 1
, OpToken_Pres (0,1) "::", UCaseToken_Pres (0,1) "Int"
]
```

From the new token list and the old presentation, an edit operation on the presentation level can be derived:

```
insert_Lay ' '
↦
replace_Pres [ LCaseToken_Pres (1,0) "simple1" ]
by [ LCaseToken_Pres (1,0) "simple", IntToken (1,0) 1 ]
```

We use an informal notation for both the *insert* and the *replace* edit operations to improve readability. The actual insert operation also contains a reference to the target location of the inserted character, and the replace operation contains the locations of the token lists, rather than the lists themselves.

2.4.4 Presentation layer: Parser

It is difficult to directly map edit operations on the presentation to edit operations on the enriched document, therefore we take the indirect approach: the edit operations are applied to the presentation, which is then parsed. The edit operations on the enriched document are computed from the new enriched document.

Similar to the scanner, the parser component of the presentation layer makes a distinction between token lists and the rest of the presentation. Only the token lists are actually parsed, the other parts of the presentation may not be edited at presentation level and are therefore recursively mapped onto their originating enriched document structures. Because parsed presentations may contain parts that are not parse, and vice versa, the two processes alternate. ***

* * * it could also be possible to parse something that is not a token list

Each part of the presentation that is not parsed is mapped directly onto the enriched document element of which it is the presentation. Because the presentation is specified with an attribute grammar, for each subtree in the presentation, the element in the enriched document that gave rise to it can be determined.*** If the originating enriched document element has children that are also presented, these children are determined via the same process. A child that does appear in the presentation is reused from the previous enriched document level, or if is not possible, it is initialized to a default value. If a child has a token list presentation, the parser process takes over.

* * * is this entirely right?

Parsing takes place by invoking a parser, which is specified in the parsing sheet, on the list of tokens. Parse errors are represented in the document with special error nodes that may appear anywhere in the document tree. *** Because the enriched document may contain information that is not presented, the parser tries to reuse the enriched document nodes from the previous time the enriched document was presented. If a node cannot be reused, the extra information is initialized to a default value. If a structural node is encountered, the previously described mapping process is invoked again.

* * * error corr. parser, with errors in local state?

Even though it is possible to specify a presentation for which it is not possible to automatically determine the originating enriched document elements, this is not a big problem in the editor. In such a case, automatic handling of presentation editing is simply not supported, and the editor designer will have to take care of handling it, or prohibit it altogether. ***

* * * what about formatters, these should probably also have tokens

Example: For the example translation at the parser component, we consider a delete operation on a number of succesive tokens in the presentation. Due to the simplicity of the example presentation, each declaration or type declaration is presented with a separate token list. Hence, presentation edit operations are always local to a single declaration or type declaration and cannot span several declarations. Therefore, the resulting edit behavior for the example is somewhat restrictive. However, in an actual source editor, the the token lists may be concatenated.

The presentation level for the example is the same as in Section 2.3.2. The part that is affected by the edit operation is:

```
...
Tokens_Pres [ LCaseToken_Pres (1,0) "simple1", OpToken_Pres (0,1) "="
            , LCaseToken_Pres (1,2) "if", UCaseToken_Pres (0,1) "True"
            , LCaseToken_Pres (0,1) "then", IntToken_Pres (0,1) "1"
            , LowercaseToken_Pres (1,10) "else", IntToken_Pres (0,1) "0"
            ]
...
```

The delete operation removes the `"if"`, `"True"`, `"then"`, `"1"`, and `"else"` tokens, giving rise to a new presentation level.

```
...
Tokens_Pres [ LCaseToken_Pres (1,0) "simple1", OpToken_Pres (0,1) "="
            , IntToken_Pres (0,1) "0"
            ]
...
```

Now, a parser is invoked on the new list of tokens. The result is not an entire enriched document, but only an updated declaration for `simple1`. For the resulting declaration, the previously presented declaration is reused. This is why the declaration contains the string `"info"`, even though it is not present at the presentation level. The string is part of the local state of the enriched document level.

```
Decl_Enr "simple1" (IntExp_Enr 0) "info"
```

The edit operation is extracted from the new enriched document part and the previous enriched document, yielding:

```
delete_Pres"if" ... "else"
↦
replace_Enr (IfExp_Enr (BoolExp_Enr True) (IntExp_Enr 1) (IntExp_Enr 0))
by (IntExp_Enr 0)
```

It should ne noted that the second `(IntExp`$_{Enr}$ `0)`in the replacement operation is exactly the same as the one in the else part of the if expression. In this case, the expression is uniquely determined by its presentation, and hence parsing it gives an exact copy, but even if this were not the case, both expressions would be the same, due to the reuse strategy of the parser. For example, if the int expression has an extra field that is not presented, the replacement int expression gets the same value for that field, since the first int expression is reused by the parser. This is analogous to the `"info"` string for the declaration.

After it is edited, an enriched document may be inconsistent with regard to the document and the evaluator. For example, if the declaration is edited in such a way that its type no longer matches the one in its type declaration. The consistency is guaranteed again when the reducer and evaluator have been invoked.

2.4.5 Evaluation layer: Reducer

The reducer takes care of mapping edit operations in derived structures on edit operation on the document. The specification of the mapping is in the *reduction sheet*, which in many cases may be automatically derived from the evaluation sheet. Automatic reduction behaviour is typically possible for parts of the enriched document that are duplicated, reordered, or partial versions of parts of the document.

The reducer resolves duplicates in the enriched document by taking the duplicate that was edited. In case of a conflict, either the edit operation may be blocked, or a choice between the two may be made. Reordering and partiality *** is handled by maintaining a mapping between each enriched document element and the document element from which it originated.

*** word: partiality?

For many derived structures, a reverse mapping does not make much sense. For example, it is hard to give a clear semantics of editing a chapter number directly, however, when two chapters in a table of contents are swapped, swapping the two actual chapters in the document could be a logical operation in some editors. In the cases for which the reverse mapping makes sense, it be specified in the reduction sheet. In the other cases, editing derived structures can be forbidden.

Besides regarding the reduction as the reverse of the evaluation, it is also possible to use reduction as an extension of the parser. For example, when a program source contains definitions of infix operators with a user-specified associativity and precedence. Parsing such operators in one pass requires a sophisticated parser, whereas the two pass solution is straightforward.

Another application of reduction is the handling of rendundancy in a document presentation. For example, when a document type for expressions does not have a construct for explicitly representing parentheses, redundant parentheses that are entered by a user can be removed by the reducer, to be added again by the evaluator. ***

*** also mention ambiguity? eg. 1:2:3:[] vs [1,2,3]?

Example:

For the reducer example, assume that the editor supports editing on an identifier in a type declaration, leading to an update on the indentifier in the corresponding declaration. Although this may not be desirable behavior in an actual source editor, it provides a good example of editing on derived values.

The enriched document edit operation that is translated is an update on the identifier in the type declaration of the enriched document from Section 2.3.1. The identifier is changed from `"simple1"` to `"simple"`. Thus, we get the following updated enriched document.

```
Root_Enr [ Comment_Enr [ "This", "is", "a", "simple", "expression" ]
         , TypeDecl_Enr "simple" IntType_Enr
         , Decl_Enr "simple1"
                    (IfExp_Enr (BoolExp_Enr True) (IntExp_Enr 1) (IntExp_Enr 0))
                    "info"
```

```
        ]
```

The reducer processes the enriched document and maps both the type declaration and the declaration onto a document level declaration. The type declaration is mapped onto a document level declaration with string `"simple"` and expression (`if True then 1 else 0`). The declaration on the other hand, is mapped on a document declaration that contains the same expression but has string `"simple1"`. The two conflicting declarations are resolved in favor of the updated fields. In this case, it means that the string `"simple"` is chosen. The result is a new document.

```
Root_Doc [ Comment_Doc ["This", "is", "a", "simple", "expression"]
        , Decl_Doc "simple"
                   (IfExp_Doc (BoolExp_Doc True) (IntExp_Doc 1) (IntExp_Doc 0))
                   "info"
        ]
```

And the document level edit operation is:

$replace_{Enr}$ `"simple1"` *by* `"simple"` $\mapsto$ $replace_{Doc}$ `"simple1"` *by* `"simple"`

2.5 Local state on each level

Each level in the Proxima editor is part of the total editor state, rather than just an intermediate value in a computation. One reason for this is to support incrementality, but a more fundamental reason is that the presentation and translation mappings are not complete. More specifically, a level does not always contain enough information to compute the level below it and vice versa. By storing both layers, together with information on how elements in each layer depend on each other, it is possible to compute the lower level from an updated upper level and vice versa.

An example of a partial presentation mapping can be found between the enriched document and the presentation. An enriched document element that is presented with tokens, does not contain the layout information that is present in the tokens. Therefore, if the element is re-presented, it must reuse the tokens from its previous presentation. In order to be able to reuse the tokens, the element must keep track of which parts of the presentation it is mapped on.

The translation of edit operations also has to deal with partial mappings. Take for example an enriched document subtree that contains only the titles of the chapters, sections, and subsections of a document. This structure does not contain enough information to construct the document. Therefore, if a title in the enriched document is edited, and we want to be able to perform the title update in the document, each enriched document node needs to keep track of the document node from which it originated.

In short, the presentation function that is given by the editor designer, specifies a mapping between the two levels *high* and a *low*: $present :: high \rightarrow low$. However, because the lower level may contain extra information, the mapping that needs to be maintained by the editor is $present :: high + extra_{high} \rightarrow low + extra_{low}$. Similarly, for the translation, the specified mapping function is $translate :: low \rightarrow high$, whereas the editor needs the mapping $translate :: low + extra_{low} \rightarrow high + extra_{high}$.

The problem with partial mappings, ie. that when mapping one level onto another, information from a previous mapping must be reused, appears in several layers of the Proxima editor. In the lower layers, the mappings can be maintained by the system, because the lower layers operate between levels that have fixed types, and the presentation and translation mappings at these layers are less customizable by the editor designer. For the higher levels, however, the editor designer in some cases needs to specify how the mappings should be maintained. The formalisms for the sheets on these levels offer special support for making this more easy. Furthermore, for frequently appearing patterns in the presentation process, special functionality is present for automatic maintenance of the mappings. Chapter **??** deals more extensively with the problem.

2.6 Motivation for the layer structure

The choice of layers for the Proxima editor is not an exact one. On the one extreme, the entire edit process can be put in one big layer, whereas on the other extreme a separate layer can be defined for every small step in the process. The choice of layers in Proxima is a balance between these extremes. This section contains the motivation for the layer structure as it is.

The separation of document evaluation from the presentation serves two purposes. Firstly, the separation makes it possible to have different presentations for a document together with its derived structures. And secondly, it facilitates the specification of edit behavior on derived structures. When parsing and reduction are mixed, such behavior is harder to specify.

The reasons for a separate layout layer are automatic whitespace handling for token presentations and efficiency, since first scanning and then parsing is more efficient than parsing on character basis. A downside is that different languages require slightly different scanning methods, and a generic scanner that is able to handle all exotic cases is hard to construct. Moreover, in some cases, an editor designer may be interested in dealing with whitespace at enriched document or document level. However, in these cases, the scanner layer may be bypassed altogether, allowing the editor designer to parse on a character basis and explicitly deal with whitespace.

Below the layout level are the arrangement level and the rendering level. The separation between these two levels and the higher levels is obvious, since the position and size computations are similar for different elements in the layout, and keeping the computations in a separate layer prevents cluttering the higher layers. The reason why the arrangement and

rendering are split is to keep the part of the architecture that deals with the GUI-specific issues as small as possible. Furthermore, the arrangement contains exact information on the location and size of each item that is to be rendered, which is useful for resolving pointing issues and performing incremental updates.

The arrangement process itself also consists of steps, since formatters are mapped on arrangement rows and columns which are then arranged similar to other rows and columns. However, these steps are very closely intertwined, and separating the arrangement into different layers does not offer enough advantages.

Besides the current layers, several other layers are imaginable. For example, a post-arrangement layer could process the arrangement in order to handle footnotes and formatting of paragraphs that contain text in languages with different reading directions. Similarly, an extra evaluation layer is conceivable when computations over computed structures need to be specified. When yet other computations are desired on the resulting computed structures, even multiple evaluation layers may be required. This brings up the issue of using higher-order attribute grammars for the evaluation layer. In fact, the whole presentation process can be specified with a higher-order attribute grammar. However, when using higher-order attribute grammars, it is not straightforward anymore to handle local state at intermediate levels, nor is it clear how to translate the edit operations on lower levels. In Proxima, these problems are dealt with by the layered architecture. Before it is possible to do presentation, or even just evaluation, with higher-order attribute grammers in Proxima, more research is necessary.