Proxima 2.0: WYSIWYG generic editing for Web 2.0

dr Martijn M. Schrage
Dept. of Computing Sciences, Utrecht University
martijn@cs.uu.nl

Abstract
In line with the Web 2.0 trend, an increasing number of web sites offer visitors the possibility to modify and add content. However, the editors provided are either basic text editors that are awkward for editing complex presentations, or custom-made editors for a specific type of content. A generic editor could be used to create powerful editors with little effort, but has the problem that it needs to be installed on each d.

A server-based generic editor can solve this problem. It will allow the easy creation of WYSIWYG editors on web-pages, without requiring the editing user to install any software. Instead, the browser runs a simple script that draws a rendering of the edited content and sends edit events back to the server.

The Proxima generic editor system has a layered architecture that can be modified in a straightforward way to support a client-server model. Because of the powerful presentation language of Proxima, web-based WYSIWYG editors for a wide range of document types can be created with little effort.

What the NLnet contributions will be used for
The NLnet contribution will be used to realize the web-based client-server version of Proxima. Furthermore, a number of interesting editors will be instantiated to showcase the capabilities of the system, thus helping to attract the attention of possible contributors to the project.

Comparison with other projects
The project can be compared with two kinds of existing projects: web-based editors and generic editors.

A multitude of web-based editors exist. Examples are Wiki's, the Yahoo! Pipes editor, and Google Docs suite. However, these editors are either mainly textual or WYSIWYG but targeted at a single type of document. No existing project offers generic web-based editing.

There are not many generic editing projects that are still being maintained. One example is Citrus (http://www.cs.cmu.edu/ NatProg/citrus.html). On the other hand, also XML editors can be regarded as generic editors, but the existing XML editors that are powerful enough to create interesting WYSIWYG editors, such as XMetaL and XMLSpy, have no support for creating web-based versions of these editors, and neither does Citrus.

The Proxima 2.0 project will be unique in the sense that it will combine WYSIWYG generic editing with web-based editing.

1 Introduction

With the advance of Web 2.0, an increasing number of web sites offer visitors the possibility to create content by editing the content of the visited page. An example is Wikipedia, which allows anyone visiting the site to edit its content.

Ideally, a visitor should be able to modify and add content in a WYSIWYG fashion, meaning that during editing, the content looks the same way as when simply viewing it. But unfortunately this is often not the case. For example, in Wikipedia, the editor offers a plain text view of the edited content. Hence, text markup such as bold and italic styles are shown as tags in the editor. Moreover, a graphical entity such as a mathematical formula is shown as text as well. For example, the simple formula $b^2 = a^2 + ab$ needs to be edited as:

```
''b''<sup>2</sup> = ''a''<sup>2</sup> + ''ab''
```

Buttons are available for inserting the text for certain tags, but these work purely textual. Hence it is possible to insert a new tag in the text of another tag, invalidating the structure. Furthermore, when an invalid structure arises, the editor offers no support to fix it or point out where the problem is.

Some sites provide more advanced editors. Take for example the editor for Yahoo! Pipes[1], which provides a graphical interface for creating web-based applications using web services from various sources (so-called mash-ups). Structures can be created by dragging components onto a canvas, editing their properties, and connecting their input and output points. Though powerful, such custom-made editors require a significant engineering effort, which puts them out of reach for many applications.

Another problem with custom-made editors is that it is difficult to combine them. If a Wiki site makes use of plug-ins which come with their own specific editor, it is not possible in general to combine the plug-in editors into a single editor. Instead, a page will be a collection of editors, each with their own specific edit model, undo buffer, etc.

The problem of building editors can be tackled by using a generic editor. When provided with the type of the edited content together with a description of its presentation and the edit behavior, a generic editor produces an editor for that specific type of document. However, this solution would require the installation of the generic editor on each device that is used to edit content, which is a serious disadvantage.

Because Internet connections are getting faster and more available quickly, a better solution to this problem is to use a web-based generic editor. The editor runs on a server and communicates with a lightweight client-script that draws a rendering of the edited content on the browser page while sending back edit events.

The Proxima editor[2] is a generic editor with an architecture that is very well suited to support a client-server model of editing. The resulting web-based editor can provide WYSIWYG editing functionality for web pages with only a fraction of the effort required to build a custom-made editor. Moreover, editors for different kinds of content can easily be mixed. Hence, we can provide edit functionality similar to a textual Wiki, but with a much richer underlying document type. Text can have visual markup, such as different fonts and colors, and editable graphical entities such as formulas can be mixed with the textual content. The client only runs a light-weight script, so no plug-in needs to be installed. Lastly, the editors will have little processing impact on the device that is used to edit the page.

[1] Yahoo! Pipes: `http://pipes.yahoo.com`

[2] Proxima homepage: `http://www.cs.uu.nl/research/projects/proxima`

The migration from desktop applications to server based applications is already taking place in other areas. E-mail clients are a good example. The first web interfaces were rather static, but more recent Ajax interfaces such as GMail and Yahoo! allow access to e-mail and folders comparable to stand-alone mail clients. Although a mail client requires less communication than an editor, applications of Virtual Network Computing show that server based applications that let all processing be done by the server are feasible. Moreover, since a generic editor need not simply send a bitmap but can send a much more efficient incremental update, the speed will be much higher than when a VNC protocol is used for editing.

Our short-term goal is to showcase the potential of the Proxima technology in order to attract the attention of possible contributors. Proxima is an open-ended system: new object types and presentations can be added to enable the creation of increasingly powerful document types. The hope is that eventually a vibrant community will contribute to a shared repository of resources and push the development of Proxima beyond what a small academia-based group can hope to achieve by itself.

1.1 Proxima 1.0

Proxima is a generic editor system developed at Utrecht University. A generic editor is a system that can be used to create editors. To create, or *instantiate*, an editor, Proxima needs to be provided with a document type and a number of sheets that describe the presentation of the document and the edit behavior. The content of a Proxima document can be mixed text, images and diagrams. A key feature of Proxima is that the instantiated editors are presentation-oriented. This means that the user performs the edit operations on the WYSIWYG screen presentation of the document. At the same time, edit operations on the document structure (such as changing a section to a subsection) are also possible. The system is written in the functional language Haskell and consists of about 15000 lines of code.

An example of an editor for a sophisticated document type created this way is the Dazzle documentation editor, an editor for the documentation of Bayesian networks. This editor combines graph editing with word-processor functionality. A document contains a Bayesian network (which is essentially a graph) and a list of sections, which contain a view on small part of the graph together with the documentation of that part.

Figure 1 shows a screenshot of the editor. The graph in the figure is an editable subgraph of the main network, so any changes to the edges are also applied to the main network, thereby solving the major problem of maintaining consistency. In the text, tags can be added to denote a reference to a network node. The text also contains references to sections and figures, the numbers of which are automatically computed. Missing references are signaled with squiggly lines.

Another example of an editor that has been obtained from generic Proxima by instantiation is a source code editor for the functional language Helium (see Figure 2). The editor shows parse and type errors in place, and allows for graphical presentations of language constructs. Structural edit operations can be combined with free-text editing. Hence, we can use structural edit operations to delete and insert list elements, which automatically removes and adds the necessary commas (or other separators). But at the same time, it is possible to delete the substring "`+2, `" from the expression "`[ 1+2, 3]`" (which does not correspond to a structural edit operation), resulting in "`[ 13 ]`".

An important aspect of Proxima is the fact that a presentation may contain computed values and structures. This is apparent in Figure 2 on the line before the declaration of `c`. The value `16` is

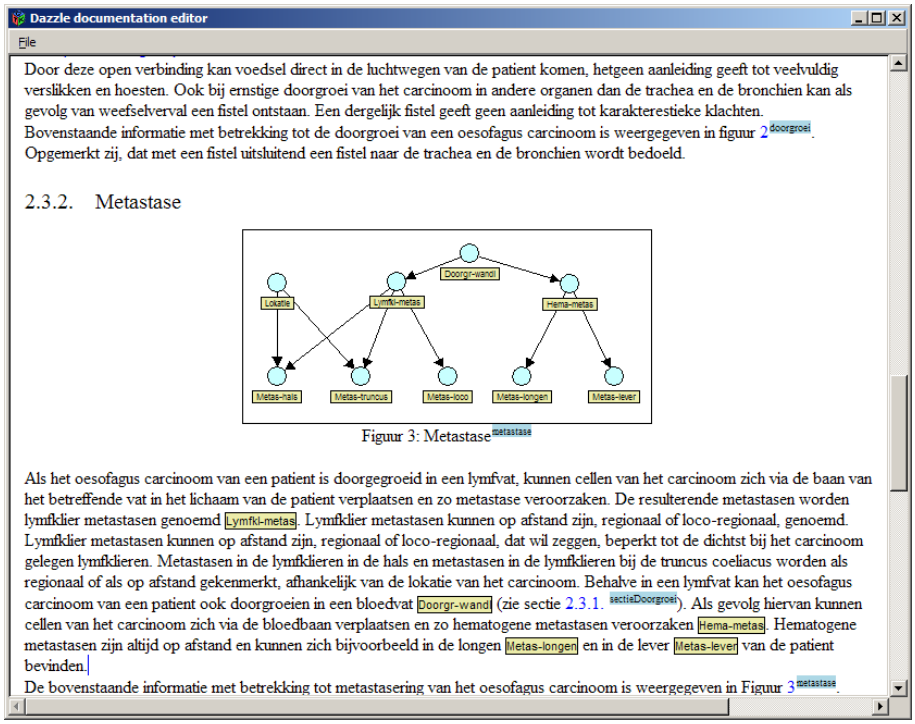

Figure 1: The Bayesian network documentation editor.

computed automatically, and changes when parameter 3 is edited or when the function `f` is modified.

Besides these two editors a multitude of editors can be implemented with Proxima. One can think of equation editors, spreadsheets, or text editors for an abbreviated syntax for XML standards such as XML Schema and XSLT, but also more exotic applications, such as a chess board, a family-tree editor, or an on-line multiple choice quiz that keeps track of the score.

1.2 The Proxima architecture

The core architecture of Proxima consists of a number of layers, each communicating with its direct neighbors. The layered structure is based on the staged nature of the *presentation* process and its inverse, the *interpretation* process. The positions at which the document, the rendering, and the intermediate data structures reside are called *levels*. Between each pair of levels we have a *layer* that maintains the mappings between its adjacent levels. Each layer consists of a presentation component and an interpretation component and may be parameterized by a *sheet*. Figure 3 schematically shows the levels and layers of Proxima. From a document type definition, a code generator generates a number of Haskell modules, which are compiled together with the sheets and the Proxima base modules to yield an editor.

A data level in Proxima is not simply an intermediate value in the presentation computation. It is an entity in its own right and maintains part of the state of the editor. The six levels of Proxima are:

- **Document:** The document structure.
- **Enriched Document:** The document attributed with derived values and structures, such as the type of a function or a table of contents.

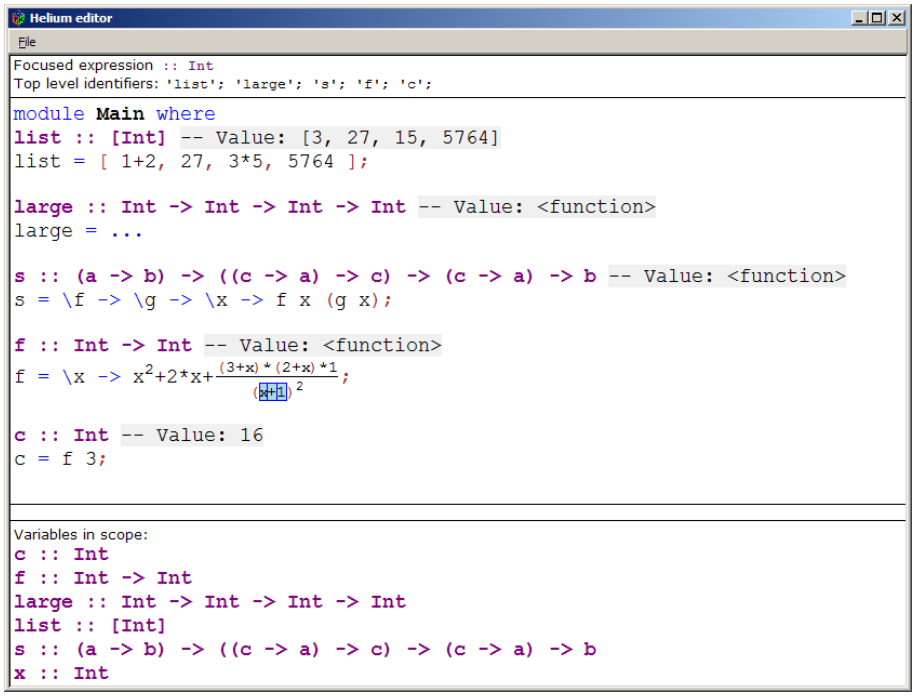

Figure 2: The Helium editor.

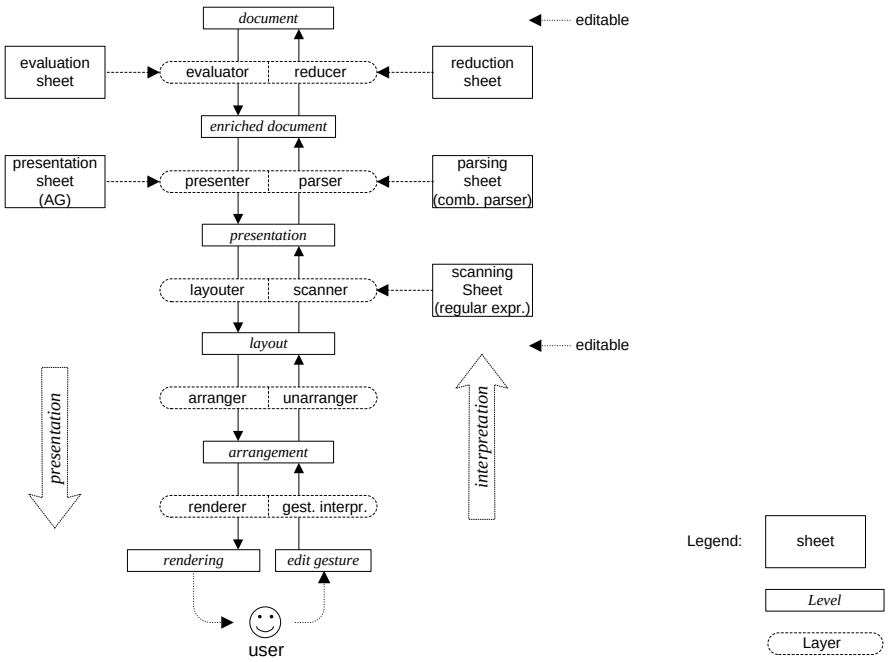

Figure 3: The levels and Layers of Proxima.

- **Presentation:** A logical description of the presentation of the document, consisting of rows and columns of presentation elements with attributes. The presentation also supports formatting based on available space (e.g. line breaking).
- **Layout:** A presentation with explicit white space.
- **Arrangement:** A formatted layout with absolute size and position information.
- **Rendering:** A bitmap of the arrangement.

After an edit operation on the document, all levels from document to rendering are updated to reflect the update. After an edit operation on the layout level, the modified layout is scanned, parsed and reduced, to obtain the corresponding updated document, from which an updated rendering is computed. Scanning and parsing does not occur after every presentation edit operation. Depending on the editor, it may occur either on a navigation operation, after a certain time interval, or at an explicit request by the user.

In order to instantiate an editor, a number of so called *sheets* must be provided:

- **Document type definition:** A EBNF grammar representation of the document type. It is similar to a monomorphic Haskell data type or an XML Schema, with support for lists.
- **Presentation sheet:** An attribute grammar that specifies for each language construct how it is presented. Simple computations, such as section counters, or static checks on the document (e.g. whether a reference is defined) can be implemented as well.
- **Scanning sheet:** A set of regular expressions for the tokens in the textual parts of the presentation.
- **Parsing sheet:** A module that contains the parsers for each part of the presentation that is presented textually.

For most editors, the computations can be specified in the presentation sheet. However, for more complex editors, two extra sheets can be provided to deal with derived structures and values: the *evaluation sheet* and its inverse, the *reduction sheet*.

- **Evaluation sheet:** Can be used to specify derived structures, such as a table of contents, or a table whose structure depends on information in the document.
- **Reduction sheet:**
  The reduction sheet specifies how updates on derived structures are mapped back onto the document. In many cases, derived values are not editable, in which case the reduction sheet simply ignores the value. However, in some cases, updates on a derived structure correspond to logical updates on the document. One can think of editing the title in a table of contents, or swapping two section titles, which causes the actual sections to swap as well.

1.3 Proxima 2.0

Turning Proxima into a web-based editor will not have a large impact on the overall implementation. Because of the strict separation between layers, which is enforced by the architecture, only four relatively small modules contain GUI-specific code. When the system needed to be ported to a new GUI library, this only took a couple of days. Hence, implementing support for an external renderer will be a relatively straightforward operation. In fact, an early version of Proxima actually used a stand-alone renderer client that it communicated with through a socket connection.

In order to realize the web-based functionality, three components need to be developed:

- An HTTP server that sends a rendering to a client and retrieves the edit commands.

- A Proxima rendering module that renders the arrangement to an XML string that is handled by the client.
- An Ajax client that renders a presentation on the screen and captures edit events, which are sent to the server.

1.3.1 The HTTP server

A simple HTTP server will handle incoming requests and start a Proxima session for each new request. The Proxima session is quite similar to an ordinary Proxima session, except that instead of catching edit events directly and drawing the rendering updates on the screen, the events are received from a socket connection, and updates are sent over this socket. This functionality can be added to the current GUI module, which contains the event loop that handles events and draws the rendering onto the screen.

1.3.2 The web-based renderer

The current renderer creates drawing commands for the GTK windowing toolkit. It traverses the arrangement data structure, and produces commands for changed parts of the arrangement that are currently in view. The new renderer will be very similar but instead of a list of commands, it will produce an XML string.

1.3.3 The Ajax client

The main implementation effort of the project will be the Ajax client. It will capture mouse and keyboard events and send these to the Proxima server. In response, it will receive an XML structure that corresponds to the update on the rendering. Textual parts of the rendering, as well as images, can be mapped onto DHTML elements. Other graphical elements, such as lines, need to be drawn separately.

1.4 Usability

One must distinguish between two levels of users of the web-based Proxima technology. One level are the people who generate specific Proxima instantiations by supplying the required sheets (in particular document type definition and presentation sheet). Except for very simple instantiations, creating these wholly anew requires considerable skills. However, as the repository of sheets grows, it will become increasingly possible to cobble new instantiations together by combining pre-existing elements. Dedicated sheet editors (themselves instantiations of Proxima) can be created to assist in this task. In any case, even with the very limited current repository, generating an instantiation is definitely much less effort than creating an equivalent editor de novo.

The other level of users consists of the visitors of websites who use a site-specific instantiation of Proxima to edit content on web pages. Here the usability will depend on the specifics of the instantiation and the complexity of the application. Proxima offers powerful possibilities for support of the edit task, as explained above. (Of course, there is no guarantee that an instantiation will actually make use of these.)

The usability of a web-based editor can be adversely affected by large delays in updates. Because Proxima has support for incrementality, only rendering updates are sent over the socket, rather than the entire rendering. Moreover, only for those parts of the rendering that are visible on screen, updates are transmitted. Therefore, most edit operations require only a very small amount of communication. Experiments with a simple web-based text editor have shown response times to be more than adequate over a wireless ADSL connection, even when several kilobytes of data were transmitted. Hence, latency is not expected to pose serious problems.