# Cobweb : A COnstraint-Based WEB browser

**Nathan Hurst** **Kim Marriott** **Peter Moulder**

School of Computer Science and Software Engineering
Monash University
Clayton, Victoria 3168, Australia
{njh,marriott,pmoulder}@mail.csse.monash.edu.au

## Abstract

We describe and evaluate Cobweb, which we believe is the first web page renderer based on constraint-solving technology. One-way constraints are used to specify position and style attributes of the document elements. Our implementation demonstrates how constraint solving provides a unifying technology for the rendering of web pages. It also demonstrates that constraint solving inherently supports viewer interaction with the document and that constraints allows more flexible and powerful specification of document layout, in particular complex text-flow, than is available with existing technologies.

*Keywords:* Web browser, adaptive documents, layout, constraint-solving, one-way constraints

## 1 Introduction

An important requirement for the web is that a web document's appearance can be adapted to its viewing environment. The same document may be accessed using a disparate variety of viewing devices and thus its appearance needs to be modified to take into account the viewing device capabilities and characteristics, such as browser window size, and also the viewer's desires and needs, such as for large fonts and language preference. Style sheet standards such as CSS (Bos, Lie, Lilley & Jacobs 1998) and XSL (Clark & Deach 1998) provide this capability, but with a large number of seemingly ad hoc layout directives.

Recent research suggests that constraints provide a powerful yet simple formalism for specifying and formalising such adaptive layout of web documents (Borning, Lin & Marriott 1997, Badros, Borning, Marriott & Stuckey 1999). Constraints can be used to simplify existing standards by allowing a precise specification of how elements, such as tables, should behave and also naturally generalise the seemingly somewhat ad hoc layout features provided in style sheet languages. But, perhaps even more importantly, we believe that constraint-solving technology provides a unifying technology for web browser implementation. The hope is that by using constraint solvers, browser implementation effort can be reduced yet provide faster readjustment of layout during viewer interaction. Here we describe and evaluate Cobweb, the first web page renderer based on constraint solving technology.

The core of Cobweb is a constraint-based text and graphics renderer called Tarpaulin. This employs a one-way constraint solver called Medieval to compute position and style attributes of the document elements. One major advantage of using a constraint solver is that they inherently support interaction: if the browser window is resized or the font size is changed by the viewer then the underlying constraint solver automatically propagates the changes to the attributes of other objects in the document. Medieval is a specialised library for doing this efficiently. Importantly, this allows Tarpaulin to support considerably more user interaction and manipulation of document elements than is possible with most browsers. For example, the user can perform direct manipulation to change the width of columns in a table, much as one can manipulate the width of columns in a spreadsheet.

Input to Tarpaulin is a description of the document in the XML-specified language, PatchML. PatchML is a generic language for specifying the text, fonts and images which make up a document and the layout relationships, declaratively specified as constraints, between them. One novel feature of PatchML is that it provides high-level constraints for specifying text flow around and within arbitrary Bézier curves (or polygons). For instance, it is simple to specify that text should flow through these columns and around these images. Other features of PatchML include user-specified colour schemes, and that the document has preconditions, dictating when it is appropriate, thus allowing alternative displays of the same logical information. Of course the document author need not write PatchML directly, instead the user could write an XSLT script to translate from XML specified languages such as XHTML to PatchML.

The most closely related research is our earlier work on the use of constraints for web page layout (Borning et al. 1997) and a constraint extensions to Cascading Style Sheets (Badros et al. 1999). The system detailed in (Borning et al. 1997) allowed the web page author to construct a document composed of graphic objects and text. The layout of these objects and the text font size were described in a separate "layout sheet" using linear arithmetic constraints and finite domain constraints. Our constraint extensions to Cascading Style Sheets, CCSS, demonstrated how CSS can be understood in terms of constraints, and how they add expressiveness. And in (Badros, Tirtowidjojo, Marriott, Meyer, Portnoy & Borning 2001, Marriott, Meyer & Tardif 2002) we have investigated constraint extensions to the Scalable Vector Graphics (SVG) web standard for vector-based graphics.

Copyright ©2003, Australian Computer Society, Inc. This paper appeared at the Twenty-Sixth Australasian Computer Science Conference (ACSC2003), Adelaide, Australia. Conferences in Research and Practice in Information Technology, Vol. 16. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

We also note the work of (Weitzman & Wittenburg 1993, Weitzman & Wittenburg 1994) who have investigated the use of relational grammars for document design. However, their interest is in specifying and recognizing layout styles rather than web browser implementation.

Another project called Madeus has used constraint solving for multimedia documents (Jourdan, Layaida & Sabry-Ismail 1997, Tardif, Bes & Roisin 2000). Madeus provides support for both temporal and spatial relationships, and it includes a rudimentary authoring environment.

There are three key differences between previous work on constraint solving for web documents and that described here. First, we have built a web page renderer from scratch rather than attempting to retrofit a constraint solver into an existing browser. This has allowed us to employ constraint-solving throughout the implementation. Second, we have provided constraints for specifying text flow in and around polygons.

Apart from the specific domain of web documents, there is a long history of using constraints in interfaces and interactive systems, beginning with Ivan Sutherland's pioneering Sketchpad system (Sutherland 1963). Constraints have also been used in several other layout applications. IDEAL (van Wyk 1982) is an early system specifically designed for page layout applications. (Harada, Witkin & Baraff 1995) describe the use of physically-based modelling for a variety of interactive modelling tasks, including page layout. GLIDE (Ryall, Marks & Shieber 1997) uses visual organization features (VOFs) to control layout of arbitrary graphs using a spring metaphor and an iterative numeric solver. Numerous systems use constraints for widget layout (Myers, Giuse, Dannenberg, Vander Zanden, Kosbie, Pervin, Mickish & Marchal 1990, Myers, McDaniel, Miller, Ferrency, Faulring, Kyle, Mickish, Klimovitski & Doane 1997), and (Badros, Nichols & Borning 2000) uses constraints for window layout.

The rest of the paper is organised as follows. In the next section we describe the Medieval constraint solver, in Section 3 we describe PatchML, and in Section 4 we describe Tarpaulin. Section 5 contains a preliminary empirical evaluation of Cobweb, and Section 6 our concluding remarks.

## 2 The Medieval Constraint Solver

Constraints are used to declaratively specify relationships between variables. An underlying *constraint solver* implicitly maintains these relationships by appropriately changing the value of other variables if one of the variables changes value.

By associating variables with the attributes of document elements we can use constraints to specify the layout and style relationships between document elements. For instance, consider the constraints

$$\begin{aligned}
&\mathit{marginsize} = 5,\\
&\mathit{margin.left} = \mathit{view.left} + \\
&\qquad \mathit{navbar.bounds.width} + \mathit{marginsize},\\
&\mathit{margin.top} = \mathit{view.top} + \mathit{marginsize},\\
&\mathit{margin.right} = \mathit{view.right} - \mathit{marginsize},\\
&\mathit{columncentre} = (\mathit{margin.left} + \mathit{margin.right})/2,\\
&\mathit{column1.right} = \mathit{columncentre} - \mathit{marginsize}/2,\\
&\mathit{column2.left} = \mathit{columncentre} + \mathit{marginsize}/2
\end{aligned}$$

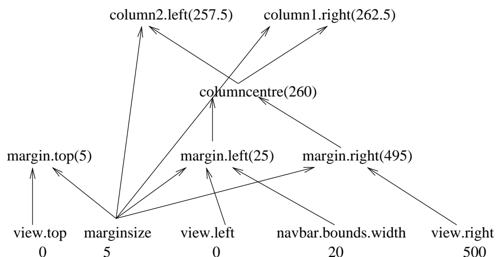
These specify that the two column rectangles are evenly spread across the available view width. The constraint solver ensures that these relationships are maintained during interaction by the viewer. Thus if the window is resized, the columns will follow.

However, there are many kinds of constraints with different classes of constraints requiring different constraint-solving techniques that vary greatly in efficiency. Much previous work on web document layout has utilised linear arithmetic constraints (Borning et al. 1997, Badros et al. 1999, Badros et al. 2001), However as discussed in (Marriott et al. 2002) one-way constraints are arguably a better choice. They are the simplest kind of constraints and have a direction: the constraints in the example above viewed as one-way constraints maintain the relationship between the variables by appropriately updating the values of, say, *columncentre* and *column2.left* (the *output variables*) whenever the other variables (the *input variables*) are changed.

The main advantages of using one-way constraints instead of linear constraints are threefold (Marriott et al. 2002):

- *Expressiveness* Linear constraints are not expressive enough. By definition they only allow us to express relations between object properties that can be written as linear functions. This is insufficient for the specification of many common types of dependency between document elements, in particular relationships involving text, such as a text box which should be just big enough to enclose some text, or relations that are based on an *if-then-else* selection. Likewise, linear constraints cannot specify dependencies of values from discrete domains, such as colours or font sizes. All of these relationships can be captured by one-way constraints.
- *Efficiency* Solving one-way constraints is considerably simpler and more efficient than solving linear constraints. (Essentially the worst case cost is linear in the number of variables.) Although not necessarily an issue for lap-top computing, this is a consideration for viewing devices, such as PDAs or mobile phones, with lower computational power.
- *Maturity* One-way constraints and constraint-solving is a very well-understood technology and one-way constraints are standard in many commercial products including widget layout in GUIs, spreadsheets, and graphic editors.

Algorithms for solving one-way constraints come in two forms: *static* for solving a system from scratch, and *incremental*, for resolving the system when some of the values change. In both cases the algorithms are couched in terms of the *constraint graph*. This is a directed graph with a node for each variable $x$ which has an associated function $f_x(y_1, ..., y_n)$ for computing the value of $x$ from the variables $y_1, ..., y_n$ and for each $y_i$ a directed edge from the node representing $y_i$ to the node representing $x$. For example, the constraint graph for the constraints given above is



The constraint graph induces a partial ordering on the variables reflecting the dependencies, i.e. variable $x$ depends on $y$ iff there is a path from $y$ to $x$. Static algorithms for solving one-way constraints first compute a total ordering for the variables which is consistent with the dependency ordering in the constraint graph, and then compute the value of the variables in this order. This can be done in time linear in the number of variables.

It is easy to modify this algorithm to be incremental. If a variable $x$'s value changes, then we first determine all variables whose value depends on $x$. Then we compute a total ordering for these dependent variables which is consistent with the dependency ordering in the constraint graph and recompute their value in this order. Again this can be done in time linear in the number of variables. For more details, the reader is referred to (Hudson 1991).

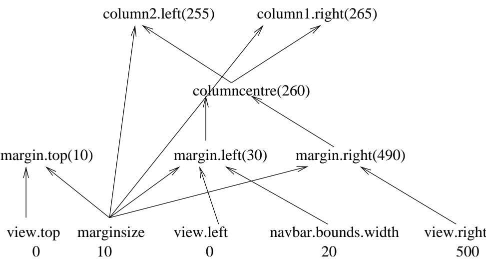
One issue is that we need to ensure that the collection of functions is well-defined in the sense that no attribute value depends recursively on itself, i.e. the constraint graph must be *acyclic*. This restriction to an acyclic constraint graph is the main drawback of one-way constraints. It means that they provide only limited support for direct manipulation. This is not an issue for Cobweb itself, although it does limit the amount of direct manipulation that can be supported by Tarpaulin, but it is an issue for PatchML authoring tools.

Another potential problem is that some constraints naturally arising in document layout are inherently cyclic. An important example is the constraint that a figure occurs on the same page that it is referred to in the text, its *anchor point*, but that it be placed before its anchor point, say at the top of the page. This is cyclic because the placement of the figure depends on the placement of the text, while the placement of the text depends on the placement of the figure since the text must flow around it.

Despite these limitations we decided that the advantages of one-way constraints outweigh their disadvantages and so used them as the basis for Cobweb. We felt it is better to start with the simplest class of constraints and extend this class when we find it necessary. Cobweb uses Medieval, our C++ implementation of a one-way constraint solver. Medieval is a very low-overhead system which allows Cobweb to use read-only variables pervasively.

Medieval provides incremental re-solving of an arbitrary directed acyclic graph defining the relationships between variables. Importantly, it also provides limited support for solving cyclic constraints. This extension allows it to solve cyclic constraints of the form identified above. The idea is that it iterates through the nodes in a cycle, computing new values for each variable, until no variable changes value, in which case a solution has been found, or a predefined iteration limit is reached (currently 5).

Medieval variables are essentially instances of classes that define a method for updating a variable's state, and which store a cache of the current value. The constraint graph shown above shows the initial state that the graph of this document might take when the page is first viewed. If the user modifies the adjustable margin widths, while viewing the page with Cobweb, Medieval will ensure that the correct values are lazily propagated up through the graph and thus that the columns remain centred on the page.



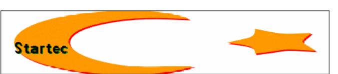

Because variables are always updated in an order which ensures that old values are not used, and that variables aren't recomputed unnecessarily, the time taken to draw a page is potentially reduced. Moreover, because the relationships are explicitly stated, the page acts in a well-defined manner when any aspect of the page differs from the author's own environment.

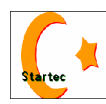
## 3 PatchML

Cobweb takes a document specified using PatchML, an instance of the XML generalised mark-up language. This is a Cobweb-specific mark-up language that provides a powerful albeit verbose description of how pages should be constructed given the environment of the browser. Of course the intent is that the author will not write PatchML directly, rather, it is intended to be directly produced by an authoring tool or by automatically translating other standards such as XHTML into it.

In essence, all elements in a PatchML document define a new variable with the name given in the `name` attribute. This variable can have multiple attributes whose values are specified using one-way constraints referring to the attributes of other document elements and some pre-defined variables capturing for instance the size of the browser window and the default font characteristics. Pre-defined variables include `view`, a rectangle describing the currently visible portion of the page; and `basefontsize`, the default fontsize for representing pages. Each element's attributes can be accessed in other definitions with a C-like syntax, such as `r1.left`.

A simple company logo might have the company name, and a simple flat colour picture:




We can define this in PatchML so that it adapts to different sized spaces in the main document. The source code below provides some examples of how the one-way constraints fit into an otherwise standard XML syntax.

```
<var name="diagbounds" defn="view"/>
<path name="moonstar"
    d="M 60 65 C 54 65 49 67 44 64
    C 32 58 29 42 40 34 C 47 29 52 30 59 31
    C 54 22 39 24 32 29 C 18 38 18 58 32 67
    C 36 70 41 71 45 71 C 51 71 56 70 73 38
    C 51 71 56 70 73 38 Z M 73 38
    C 73 45 72 44 67 48 C 72 50 73 51 73 56
    C 81 53 77 52 86 54
    C 83 47 82 49 85 41 Z"/>
<plainregion name="whitebkgnd"
    bounds="rect(diagbounds.left - 2,
                 diagbounds.bottom - 2,
                 diagbounds.right - 2,
                 diagbounds.bottom -2)"/>
```

```
<plainregion name="moonshadow" fill="red">
  <poly src="moonstar"/>
</plainregion>
<plainregion name="logo"
  bounds="diagbounds"
  fill="rgb(255,153,0)">
  <poly src="moonstar"/>
</plainregion>
<text name="star">startec</text>
<textregion name="starshadow" from="star"
   bounds="rect(diagbounds.left + 21,
                diagbounds.bottom - 30,
                diagbounds.right,
                diagbounds.bottom)"
   fill="green"/>
<textregion name="star" from="star"
   bounds="rect(starshadow.bounds.left-1,
         starshadow.bounds.top-1,
                starshadow.bounds.right-1,
                starshadow.bounds.bottom-1)"
   fill="black"/>
```

We now describe the elements in more detail. The simplest tag is a variable definition:

```
<var
   name="⟨string literal⟩"
   defn="⟨evaluated expression⟩"
/>
```

This connects a new variable with name given by `name` and whose value is given by the expression in `defn`. The type is inferred by the result of the expression. For example, numbers, colours and rects can be created this way.

Numeric expressions are built using the standard arithmetic operators, and functions like *abs*() for absolute value. A heading's fontsize might be defined as `<var name="headfs" defn="basefontsize * 2"/>`

Our colour specification is taken from the CSS3 Color module work-in-progress(Çelik, Lilley & Pettit 2001). Colours are constructed using the `rgb(red, green, blue)` function (taking three floating point arguments in the range $[0, 255]$), while opacity (alpha) for blending may be specified as a scalar in the range $[0, 1]$. Using one-way constraints, PatchML can define colour schemes based on a few colours and allow the end user to specify these base colours. A common colour would be defined beforehand: `<var name="white" defn="rgb(255,255,255)"/>`

PatchML is a rectilinear page layout system, like HTML or Word. That is, all layout is understood in terms of the horizontal and vertical orientation of the web page. Rotation of text and other elements is not provided. As the elements are represented with coordinate alignment, PatchML provides a `rect(left, top, right, bottom)` function which defines the axis aligned rectangle or box.

PatchML provides the author with arbitrary shapes using the `path` element. This is based on SVG paths (Ferraiolo 2001) and has an identical syntax. All paths are closed. The `d` attribute in particular, describes the path using the same form as the `d` attribute of SVG's path element.

```
<path
   name="⟨string literal⟩"
      The reference name of this element.
   d="⟨string expression⟩"
/>
```

For example, see the definition of the path `moonstar` in the running example. In our current version, coordinates in a path specification must be constants and cannot be given by expressions involving one-way constraints.

Cobweb provides functions that take paths and compute their `union`, `intersection` and `difference` to define a new path.

Rectangles are treated as a type of path: anywhere a path is expected, a rectangle expression can be given. However, this is a directed conversion operation: rectangle properties can't be specified as Bézier or polygon expressions.

Paths are used to define regions where text can appear or an image can be painted. These regions are defined as the set of points contained within any of a set of polygons. We use the positive winding number rule for inclusion.

PatchML provides three kinds of regions: `plainregion`, `imageregion`, and `textregion`. Every region has a `bounds` attribute, which is the rectangle in the viewing canvas where the region is to be placed (specified in the viewing canvas coordinates) and an optional `border`, which is a path defining the perimeter of the region. There is an implicit scaling transformation mapping the bounding box of the `border` to `bounds`.

The simplest kind of region is the plain region:

```
<plainregion
   name="⟨string literal⟩"
      The reference name of this element.
   bounds="⟨rectangle expression⟩"
      expression to compute the bounds of the element.
   fill="⟨colour expression⟩"
   opacity="⟨scalar expression⟩"
   border="⟨path expression⟩"
/>
```

This paints the specified region in a single colour and opacity. They are used for filling any area with colour. For instance, setting the background colour of a column.

Image regions take pixels in the `srcrect` in the source image and map them linearly to those in the `bounds` on the page.

```
<imageregion
   name="⟨string literal⟩"
   src="⟨string expression⟩"
   srcrect="⟨rectangle expression⟩"
   bounds="⟨rectangle expression⟩"
   border="⟨path expression⟩"
/>
```

Image files are requested through the `<image>` tag. The separation of loading from display is used to provide a logical structure for referring to the loaded image's properties.

An image is specified using the `image` element:

```
<image
   name="⟨string literal⟩"
   url="⟨string expression⟩"
/>
```

The most interesting kind of region is the `textregion`, which performs text layout. This behaves like the `plainregion`, except that lines of text

are placed where a `plainregion` would paint a colour. The text region 'chews up' text from the source text. Further text regions using the same source text thus act like CSS flows. I.e. the ordering of the text regions in the PatchML document determines the order in which text flows through them, not their position on the drawing canvas.

```
<textregion
   name="⟨string literal⟩"
   from="⟨text expression⟩"
   bounds="⟨rectangle expression⟩"
   border="⟨path expression⟩"
>
      <exclude path="⟨B"/>
</textregion>
```

The `exclude` attribute allows the designer to conveniently exclude other regions from the area that the text is drawn. This could be done by using union and difference operations for the region, but in practice it was found that this was worth the extra tag.

Text regions also provide a number of attributes for tuning column layout, including lower and upper bounds on the glyph $y$ coordinates. These are used to allow columns to be only as long as the required.

The actual stream of text content is specified using the text element:

```
<text
   name="⟨string literal⟩"
>
      CDATA
</text>
```

The text stream can include font colour, size and style attributes, much like HTML. Unlike HTML, these can be specified using arbitrary one-way constraints. Thus a designer can allow the page to have adjustable font size to be modified by the user and still give results that the designer wanted. Font faces are specified with URLs to a truetype file.

An important aspect of document layout is the ability to be able to position floating regions in terms of the placement of specific elements in a text stream. PatchML provides this ability through anchor points. An anchor point is a variable whose value is the bounding box of the glyphs of the characters contained in the `anchor` tag.

```
<anchor
   name="⟨string literal⟩"
>
      CDATA
</>
```

Anchor points provide a method of aligning diagrams and in-text image boxes with the text flow. The aligned element's region can still be used to exclude text, which makes provision of typographic features such as drop-caps significantly easier.

A major difficulty with anchors is resolving cyclic dependencies that occur when anchors affect preceding text. However, due to the slack space in text, it is often possible to modify the layout of preceding text so long as there are enough lines of text to take up the slack. Otherwise Medieval makes use of its iterative approach to solving cyclic constraints.

One-way constraints, in particular the use of *if-then-else*, gives considerable power to the designer to adjust layout and style. However, sometimes a more significant change is warranted. To support this, *alternative* PatchML layouts for a logical page are allowed. Each layout has an optional pre-condition specified using one-way constraints. This specifies whether the layout in question is suitable for the browser configuration. It is possible to include the precondition at the head of a PatchML document, or in separate linking document, which contains the ordered set of page layouts.

## 4 Tarpaulin

Tarpaulin is the core of the Cobweb browser. It reads in a PatchML document, using the `libxml` library suite to parse the XML and create an internal representation for the document. This internal representation takes the form of one-way constraints between the document elements capturing the desired layout. Tarpaulin then calls Medieval to solve these constraints and hence to compute the layout of the document and to check that the document preconditions are satisfied. Tarpaulin then renders the document to the screen. It supports simple editing of the document's appearance by the viewer.

Tarpaulin relies upon the `libart` library, the `freetype` text rendering code, and the `gdk-pixbuf` library. The `libart` library provides routines for polygon rendering and various geometric primitives. The `gdk-pixbuf` library provides code for loading images and for transferring the rendered tarpaulin to the screen while `freetype` provides the rasterization of arbitrary truetype fonts.

The document representation used by Tarpaulin mirrors the elements of the PatchML document with each kind of PatchML element having a corresponding class in Tarpaulin.

Thus for example, Tarpaulin provides a class `rect` to represent rectangles in PatchML. This represents axis aligned rectangles and provide intersection, minimal bounding box operations, a scale dimensions function which scales the width and height of a rectangle, a mapping function which takes three rectangles, *src*, *dest*, and *r*, and returns the rectangle with coordinates in *dest* which corresponds to *r* relative to *src*, an analogous function for points and methods to extraction of parts of the rectangle such as the left edge coordinate. Also provided is a `point` constraint class and several operations for manipulating `points` and `rect`.

The current implementation represents colours as a 4-tuple of red-green-blue-alpha. In future we intend to look more closely at (Çelik et al. 2001) and the works it refers to. It is possible to define colour schemes based on some parametric colours and allow the viewer to specify these base colours.

Tarpaulin represents polygon regions with Sorted Vector Paths(SVPs). In part this is because `libart` provides operations on polygons represented in this form since this allows efficient rendering and union, intersection, scale and translation operations are also very efficient. SVPs are either directly constructed from an PatchML path declaration or computed using intersection, union, set difference, and a mapping function which maps an SVP from the coordinates of a *src* rectangle to the coordinates of a *dest* rectangle.

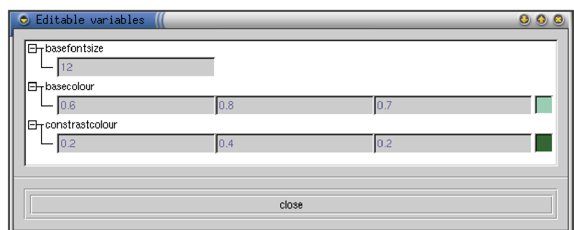
The rendering canvas is called the *tarpaulin*. The visual objects on a tarpaulin are called *patches*. All patches theoretically cover an infinite plane, but they

are only drawn within a region bounded by an SVP. There are *plain*, *image* and *text* patch types corresponding to the respective PatchML regions.

The most novel feature of Tarpaulin is that it uses constrained variables to store almost all internal state and a one-way constraint solver to compute their value. This not only includes those attributes modelling the position and appearance of document elements, but also variables storing intermediate results used to compute these attributes. Most of the operations provided by the Tarpaulin classes actually set up one-way constraints in Medieval. The main advantage of this is that the constraint solver provides automatic, efficient re-computation of all attributes if any variable is changed. In particular, attributes of patches can be specified as functions of the width and height of the viewing region `view`. This means that the document layout will be recomputed automatically as the window is resized (and the relevant Gtk widget gets resized). This is in contrast to existing browsers in which (re)computation of attributes must be explicitly programmed by the browser writer.

In addition to resizing the browser window, constraint solving also allows efficient support for other forms of interaction by the document viewer. As we have indicated one limitation of one-way constraints is that they must be acyclic

The document designer specifies which variables in a PatchML document are editable by the viewer and the associated control. Any root variable marked as `editable="true"` using the variable's attribute is modifiable in a list. For example, here is the dialog to set the document font size and colour preferences:




Several control widgets are also provided as part of the Tarpaulin user interface so that the document designer can give the viewer the ability to modify aspects of the document layout in a simple and consistent manner. Controls currently provided in PatchML include: a `<movablepoint>`, constrained to a polygon,

```
<movablepoint
   name="⟨string literal⟩"
   bounds="⟨rectangle expression⟩"
   border="⟨path expression⟩"
/>
```

a bounded `<slider>` which can follow a given line segment,

```
<slider
   name="⟨string literal⟩"
   lower="⟨scalar expression⟩"
   upper="⟨scalar expression⟩"
   bounds="⟨rectangle expression⟩"
/>
```

or a set of markers that specify the bounds of a rectangle:

```
<adjustablerect
   name="⟨string literal⟩"
   initialbounds="⟨rectangle expression⟩"
/>
```

These controls provide considerable power. They can be included by the designer to allow the viewer to resize diagrams, provide zoomable images, adjust the width of columns in a table (even hiding unwanted columns) or to allow the user to adjust the font size interactively. As discussed in (Marriott et al. 2002) another kind of viewer interaction that one-way constraints can support is *semantic zooming* in which the viewer can interactively change the level of detail in a document when viewing it, allowing for example the viewer to collapse some sections of a document to just a heading while expanding those parts of the document they are interested in.

As an example, see Figure 1. We have a picture of a daffodil with a slider at the bottom for zooming in on the detail in the picture. Clearly other aspects of a page can be adjusted by the slider to provide semantic zooming by hiding things as they become to small to contribute to the page.

The zoomable picture is made by connecting the slider through the one-way constraint system to the source rectangle in the image:

```
<plainregion name="whitebkgnd"
  bounds="view" fill="rgb(255,255,255)"/>
<image name="daffodil" src="daffodil.png"
<slider name="magnification"
  lower="daffodil.width" upper="daffodil.width/4"
  bounds="rect(0,view.bottom-16,
    view.right, view.bottom)"/>
<var name="cx" defn="daffodil.width/2"/>
<var name="cy" defn="daffodil.height/2"/>
<path src="circle"/>
<imageregion name="maglens"
  image="daffodil"
  srcrect="rect(cx-magnification,
    cy-magnification,
    cx+magnification,
    cy+magnification)"
  bounds="rect(0,0,view.right,
    view.bottom-16)"
  path="circle"
/>
```

## 5 Empirical Evaluation

Our preliminary experience with Cobweb is that drawing and redrawing of web pages is comparable in speed to that provided by mature web browsers. As an example, we have compared the time taken by Mozilla and Cobweb to redraw the reasonably complex page shown in Figure 2 as it is resized from a full screen window down to a half screen window. The tests were done on a 24-bit X display running on a 400MHz PowerPC-based laptop.

We timed how long it took each browser to redraw the page at 30 intermediate decreasing sizes. On average, Mozilla took 0.4 seconds while Cobweb took 0.35 seconds. We then used the profiling tool `gprof` to determine how much time was actually spent by Cobweb in drawing. We found that graphics related functions took up 75% of the total redraw time, on average. Unfortunately we couldn't perform the same analysis for Mozilla since we don't have a detailed knowledge of the functions of its internal subroutines.

Figure 1: using a slider to adjust the size of an image interactively

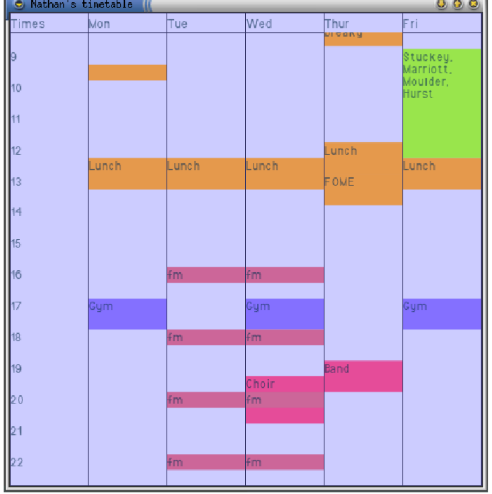



Figure 2: Web-page used to compare redraw speed of Mozilla and Cobweb

The results are quite encouraging suggesting that constraint solving takes significantly less time than graphics rendering and that the one-way constraint solver is of comparable speed to Mozilla's layout engine. Indeed we suspect it is faster since Mozilla uses more efficient graphics rendering code (though Mozilla must update a considerably more complex user interface.)

Another important aspect to evaluate is the expressiveness and power of one-way constraints. Broadly speaking they allow us to model the layout idioms provided by HTML. However, Cobweb does have some limitations. In particular, layout of structured text such as lists is not supported. On the other hand one-way constraints as provided in Cobweb are more expressive than HTML. For instance, it allows multicolumn text layout around complex images. As an example consider the web page containing example text taken from Hans Christian Andersen's story "The Happy family" in figure 3. It uses one-way constraints to spread the first few paragraphs of the story across two columns with a small picture of a dock leaf centred between the columns. The left margin is wider to include a navigation bar. It also uses constraints to specify a drop cap on the first letter in the story.

## 6 Conclusion

We have described Cobweb, to our knowledge, the first web page renderer based on constraint solving. Cobweb clearly demonstrates the advantages of employing constraint solving technology in web browsers. It shows that:

- Constraint solving provides a uniform mechanism for many apparently different parts of the browser, for instance the font relationships, the window size, column spacing and colour scheme.
- Constraints provide considerably more expressiveness than can be found in traditional web mark-up languages.
- Constraint solving naturally supports viewer interaction and efficient dynamic adjustment of layout.

Figure 3: Example of multicolumn text layout

As Cobweb is very much work in progress, there are many aspects demanding further research.

Text is still not powerful enough. It would be better to be able to treat a set of text regions as a single large column. In this column images, plain colours or glyphs can be drawn sequentially. Without this there is no way elegant way to put a heading in line with the text in a multiple column setting, or to fill multiple columns with combined text, plain patches and images.

Another limitation of Cobweb is its restriction to one-way constraints and its handling of cyclic constraints. In practice we have found that iteration isn't always powerful enough to handle tables or float placement on a page in the presence of anchor constraints since in both cases constraints with cyclic dependencies are needed. It would also be desirable to allow constraints with different strengths. We plan to add linear constraint solving capabilities and dynamic linear approximation (Hurst, Marriott & Moulder 2002) in a future implementation.

Finally, PatchML does not allow document layout to be formalised in terms of page templates which depending upon page content can be used multiple times and which allow expressions in constraints to refer to attributes of the "current" page or column. This is vital for multipage documents.

## 7 Acknowledgements

We thank Peter Stuckey for many helpful conversations in particular about the graph algorithms used for constraint solving within Medieval. We also thank David Salesin and his research team at Microsoft Research for their interest in this work.

## References

Badros, G. J., Borning, A., Marriott, K. & Stuckey, P. (1999), Constraint cascading style sheets for the web, *in* 'Proceedings of the 1999 ACM Conference on User Interface Software and Technology', ACM, New York.

Badros, G. J., Nichols, J. & Borning, A. (2000), SCWM—the Scheme Constraints Window Manager, *in* 'Proceedings of the AAAI Spring Symposium on Smart Graphics'.

Badros, G., Tirtowidjojo, J. J., Marriott, K., Meyer, B., Portnoy, W. & Borning, A. (2001), A constraint extension to scalable vector graphics, *in* 'Proceedings of WWW10'.

Borning, A., Lin, R. & Marriott, K. (1997), Constraints for the web, *in* 'Proceedings of ACM MULTIMEDIA'97'.

Bos, B., Lie, H., Lilley, C. & Jacobs, I. (1998), 'Cascading Style Sheets, level 2 CSS2 Specification. W3C Recommendation', `http://www.w3.org/TR/REC-CSS2`.

Çelik, T., Lilley, C. & Pettit, B. (2001), 'CSS3 module: Color', `http://www.w3.org/TR/2001/WD-css3-color-20010305`.

Clark, J. & Deach, S. (1998), 'Extensible Stylesheet Language (XSL), Version 1.0. World Wide Web Consortium Working Draft', `http://www.w3.org/TR/WD-xsl`.

Ferraiolo, J. (2001), 'Scalable vector graphics (svg) 1.0 specification'.

Harada, M., Witkin, A. & Baraff, D. (1995), Interactive physically-based manipulation of discrete/continuous models, *in* 'SIGGRAPH '95 Conference Proceedings', ACM, Los Angeles, pp. 199–208.

Hudson, S. (1991), 'Incremental attribute evaluation: A flexible algorithm for lazy update', *ACM Transactions on Programming Languages and Systems* **13**(3), 315–341.

Hurst, N., Marriott, K. & Moulder, P. (2002), Dynamic approximation of complex graphical constraints by linear constraints, *in* 'Proceedings of User Interface Software & Technology', ACM, New York, pp. 191–200.

Jourdan, M., Layaida, N. & Sabry-Ismail, L. (1997), MADEUS: An authoring environment for interactive multimedia documents, *in* 'International Conference on Multimedia Computing and Systems', pp. 644–645.

Marriott, K., Meyer, B. & Tardif, L. (2002), Fast and efficient client-side adaptivity for svg, *in* 'ACM Conference on the World Wide Web (WWW 2002)'.

Myers, B. A., Giuse, D. A., Dannenberg, R. B., Vander Zanden, B., Kosbie, D. S., Pervin, E., Mickish, A. & Marchal, P. (1990), 'Garnet: Comprehensive support for graphical, highly-interactive user interfaces', *IEEE Computer* **23**(11), 71–85.

Myers, B. A., McDaniel, R. G., Miller, R. C., Ferrency, A. S., Faulring, A., Kyle, B. D., Mickish, A., Klimovitski, A. & Doane, P. (1997), 'The amulet environment: New models for effective user interface software development', *IEEE Transactions on Software Engineering* **23**(6), 347–365.

Ryall, K., Marks, J. & Shieber, S. (1997), An interactive constraint-based system for drawing graphs, *in* 'Proceedings of UIST 1997', Banff, Alberta Canada.

Sutherland, I. (1963), Sketchpad: A man-machine graphical communication system, *in* 'Proceedings of the Spring Joint Computer Conference', IFIPS, pp. 329–346.

Tardif, L., Bes, F. & Roisin, C. (2000), Constraints for multimedia documents, *in* 'Proceedings of the Second International Conference and Exhibition on the Practical Application of Constraint Technology and Logic Programming'.

van Wyk, C. J. (1982), 'A high-level language for specifying pictures', *ACM Transactions on Graphics* **1**(2), 163–182.

Weitzman, L. & Wittenburg, K. (1993), Relational grammars for interactive design, *in* 'IEEE Symposium on Visual Languages', pp. 4–11.

Weitzman, L. & Wittenburg, K. (1994), Automatic generation of multimedia documents using relational grammars, *in* 'Proceedings of 2nd ACM Conference on Multimedia'.