Constraint-Based Document Layout for the Web*

Alan Borning†

Dept. of Computer Science & Engineering,

University of Washington, Seattle, Washington 98195, USA.

borning@cs.washington.edu

Richard Kuang-Hsu Lin and Kim Marriott

School of Computer Science & Software Engineering,

Monash University,

Clayton, Victoria 3168, AUSTRALIA.

{rlin,marriott}@cs.monash.edu.au

Abstract

Constraints can be used to specify *declaratively* the desired layout of a web document. We present a system architecture in which both the author and the viewer can impose page layout constraints, some required and some preferential. The final appearance of the web page is thus the result of negotiation between author and viewer, where this negotiation is carried out by solving the set of required and preferential constraints imposed by both parties. We identify two

*An earlier version of this paper entitled "Constraints for the Web" appeared in the *Proceedings of the Fifth ACM International Multimedia Conference*, pages 173–182, Seattle, November 1997

†This research was conducted while visiting Monash University.

plausible system architectures, based on different ways of dividing the work of constraint solving between web server and web client. We describe a prototype constraint-based web authoring system and viewing tool that provides linear arithmetic constraints for specifying the layout of the document as well as finite domain constraints for specifying font size relationships. Finally, we provide an empirical evaluation of the prototype.

1 Introduction

One of the main technical challenges of document delivery over the Internet results from the tensions among the document designer's desire to specify the exact appearance of the document, the document viewer's desires and needs, and the capabilities of the viewing device and browser. On the authoring side, current web authoring tools do not allow the author to specify how the document layout should change in response to the viewer's desires. And on the viewing side, web documents are often less flexible than one might like: typically the user of a web browser has only small control over the appearance of the presented information — the viewer can resize the browser or set default font information, but not much more. Constraint-based layout provides a possible solution.

A constraint is simply a statement of a relation (in the mathematical sense) that we would like to have hold. Constraints have been used for many years in interactive graphical applications for such things as specifying window and page layout, specifying relationships among parts of a drawing, specifying animations, maintaining consistency between application data and a view of that data, maintaining consistency between multiple views, and representing physical laws in simulations. They allow the designer to specify *what* are the desired properties of the system, rather than *how* these properties are to be maintained. The major advantage of using constraints is that they allow partial specification of the layout which can be combined with other partial specifications in a predictable way.

It is thus natural to consider constraint-based tools to aid in authoring web documents. We describe a system that allows web authors to employ constraints to specify page layout, including figure placement and column widths and spacing as well as relative font sizes. Some of these constraints will be requirements that must be satisfied, while others may be preferences of different strengths that can be relaxed if need be. In addition, authors can use several *constraint style sheets* to specify alternate sets of constraints to be used under different circumstances, for example, for a one versus a two-column layout. The conditions under which a style sheet is applicable are, of course, also specified as constraints.

Constraints may be imposed by the viewer as well as by the author. Like those of the author, some of the viewer's constraints can be preferences as well as requirements. The final appearance of the document is thus in effect the result of a *negotiation* between the author and the viewer — where this negotiation is carried out by solving the set of required and preferential constraints imposed by both parties and also taking into account the browser's capabilities (which may also be couched as constraints).

This negotiation model leads to two possible system architectures. In one model, both the authoring tool and the viewing tool can perform runtime constraint solving. The authoring tool uses the solver while constructing and testing the pages and applets, while the viewing tool uses a different solver (on the viewing machine) to solve the combined constraints from the author and viewer to determine the final page layout. In this case a compact representation of the constraints, along with the content of the page, additional layout information, and applets, is shipped over the network for each page. In addition, the runtime solver must either be provided as a plug-in or shipped (once) and saved on the viewer's machine.

In the other model, the authoring tool again uses a powerful runtime constraint solver, but only a restricted set of constraints is available to the viewer. The authoring tool compiles a Java program representing a plan for satisfying the author's constraints and the predetermined kinds of constraints that the viewer may impose. This program is then shipped to the viewer's machine —

so that a runtime constraint solver is not needed on the client side.

As a proof of concept we have implemented a prototype constraint-based authoring system that embodies the first architecture. It provides two types of constraints: linear arithmetic constraints for document layout in which elements are viewed as "boxes," and finite domain constraints for specifying the relationships between font sizes in the document. Our empirical evaluation shows that existing constraint-solving techniques are sufficiently fast for even quite complex documents whose layout specification involves up to 300 constraints.

The paper is organized as follows. In Section 2 we describe constraint-based web page layout and the negotiation model in more detail and demonstrate that constraints provide important functionality for web page layout. Section 3 contains an analysis of the requirements on the constraint solver, and of different ways the constraint solving responsibilities can be partitioned between client and server, while Section 4 describes the two constraint solving algorithms we employ. Section 5 describes our prototype implementation, while Section 6 contains our empirical evaluation. Finally, Sections 7 and 8 discuss related work, and conclusions and directions for future work.

2 Constraint-Based Page Layout

With current document markup languages, such as HTML 4.0, including Cascading Style Sheets (CSS 1.0 and 2.0), the layout of the page is relatively static and fixed by the designer. In principle the client has the ability to change fonts and size of fonts and to resize the document. In practice, however, this freedom is limited, since if these attributes are significantly changed from what the designer intended, the document's appearance will often be unsatisfactory. The problem is that current markup languages and their associated style sheets do not provide the designer with the capability to control precisely how the layout of the document should change if these parameters are modified.

A solution to this problem is to use constraints to specify the core aspects of the design layout. The constraints capture the "semantics" of the design, those

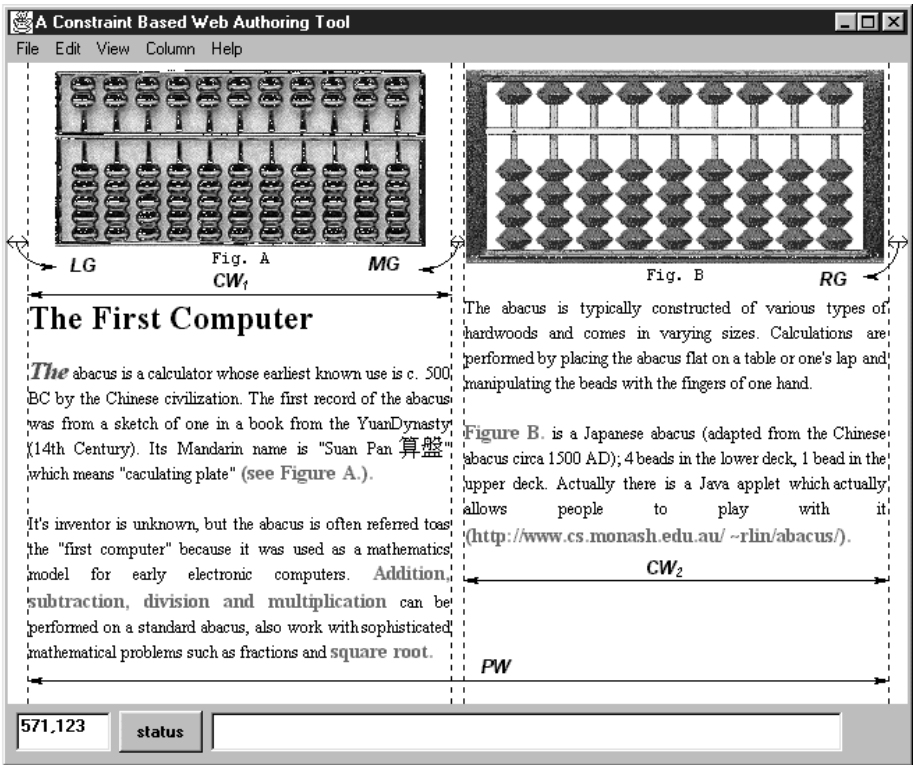

Figure 1: Two Column Layout for the Abacus Document

aspects that must hold true for the layout to be appealing. The designer can specify placement of the document elements using linear arithmetic equalities and inequalities. Such constraints allow easy specification of table, column, and image placement in a way that scales gracefully. The designer can also use constraints on font sizes to control the appearance of text, allowing the document appearance to capture the desired "look and feel" regardless of which fonts are available to the browser.

As an example, consider the page layout shown in Figure 1. We require that the text is arranged in two columns, that figures A and B are centered in the first and second columns respectively, and that the tops of the two figures are aligned horizontally. These layout constraints are captured by the following equalities and inequalities:

$$
\begin{array}{ll}
(1) & PW = LG + MG + RG + CW_1 + CW_2 \\
(2) & CW_1 = CW_2 \\
(3) & LG = RG = 0.05 \times CW_1 \\
(4) & MG = 0.7\text{cm} \\
(5) & FigA.midx = LG + 0.5 \times CW_1 \\
(6) & FigB.midx = LG + CW_1 + MG + 0.5 \times CW_2 \\
(7) & FigA.top = FigB.top \\
(8) & FigA.width \leq CW_1 \\
(9) & FigB.width \leq MG + CW_2 + RG
\end{array}
$$

Constraint (1) states that the page width, PW, is the sum of the widths of the left, middle and right gutters LG, MG, and RG, and the two columns, CW_1 and CW_2. Constraint (2) states that the two columns have equal width; (3) states that the left and right gutters are equal and are 1/20 of the width of the columns; (4) states that the middle gutter is of fixed size (0.7 cm); (5) states that the x value of the midpoint of Figure A is at the center of the first column; (6) states that Figure B is centered in the second column; while the last equality (7) enforces that the two figures are horizontally aligned. The inequalities (8) and (9) enforce that the columns are wide enough for Figures A and B.

For a given value of the page width PW we can find a solution to the other variables that satisfies these constraints and that gives us a layout. For instance, if $PW = 21.7$ cm then $LG = RG = MG = 0.5$ cm and $CW_1 = CW_2 = 10$ cm. Conversely, if $PW = 42.7$ cm then $LG = RG = 1.0$ cm, $MG = 0.7$ cm and $CW_1 = CW_2 = 20$ cm. Note how the left and right margins scale with respect to the page size while the middle gutter has an absolute size.

This model is, however, a little too simple. In particular it does not allow the designer to state preferences for values. Thus in the above example, for a given PW the equations do not constrain the vertical placement of Figures A and B. Allowing preferred values permits the designer to specify that they should be placed as closely as possible to the first reference to these figures in the text.

Preferred values also allow the designer to give default values for parameters in the layout.

We therefore extend our model to allow the user to specify that an inequality or equality is preferred but not required, so that the constraint should be satisfied when possible but does not need to be. Constraint hierarchies [5] formalize such preferences. A constraint hierarchy consists of collections of constraints each labelled with a strength. There is a distinguished strength label *required*: such constraints must be satisfied. (Actually, we will usually omit the "required" label: constraints without a label are assumed to be required.) The other strength labels denote preferences. There can be an arbitrary number of such strengths, and constraints with stronger strength labels are satisfied in preference to ones with weaker strength labels. In our example, the equalities and inequalities given earlier are required constraints, and we will use *weak*, *medium*, and *strong* to label non-required constraints. The strength of each constraint is chosen by the designer. While there can be an arbitrary number of different strengths, in practice the designer selects from a small, fixed set. Generally speaking stronger strengths are used to preserve the "structure" of an object such as the height and width of an image, while less strong constraints are used for placement. Given a system of constraints, the constraint solver must find a solution to the variables that satisfies the required constraints exactly, and that satisfies the preferred constraints as well as possible, giving priority to the stronger preferred constraints.

Preferred values for variables can be modeled as simple non-required constraints of the form $v = c$ for a variable v and constant c. Such preferred values are particularly important when specifying font sizes, since the precise font size desired by the document designer may not be available to the browser. In the above example we might add the constraints

$$\begin{array}{rl} \mathit{medium} & TextFontSize = 12\text{pt} \\ \mathit{weak} & CaptionFontSize = TextFontSize \\ \mathit{weak} & HeadingFontSize = 2.2 \times TextFontSize \end{array}$$

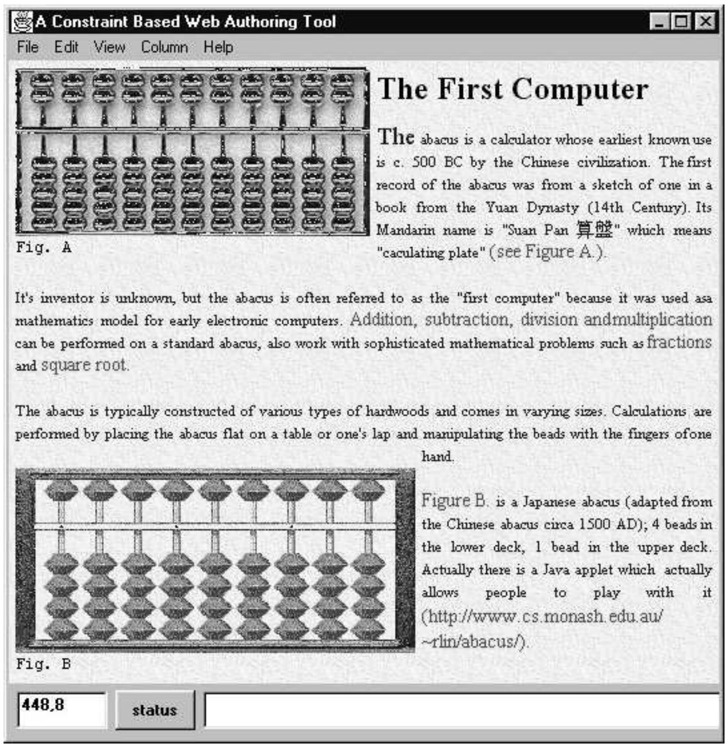

Figure 2: One Column Layout for the Abacus Document

which specify the medium preference that the text font size is 12 pt and two weak strength constraints relating the caption font size and heading font size to the text font size.

In the previous example, we have required that the columns be wide enough for Figures A and B. If they are not, then this is not an appropriate layout. For instance, if the width of Figures A and B is 10 cm, then we cannot solve the constraints when the page width is less than 21.3 cm. The obvious question now is: what happens if the constraint system is unsatisfiable for a given page width? There are two possibilities. The first is to use a scroll bar which allows the viewer to scroll over the smallest valid layout. A better solution is for the designer to provide an alternative design for the case when the page width is too small.

In this case the designer might wish to use a single column with the following constraints:

$$
\begin{array}{rl}
 & PW = LG + CW + RG \\
 & LG = RG = 0.6\text{cm} \\
 & FigA.leftx = LG \\
 & FigB.leftx = LG \\
 & FigA.width \leq CW \\
 & FigB.width \leq CW \\
 & PW \leq 26\text{cm} \\
\textit{medium} & TextFontSize = 11\text{pt} \\
\textit{weak} & CaptionFontSize = 1.1 \times TextFontSize \\
\textit{weak} & HeadingFontSize = 2 \times TextFontSize
\end{array}
$$

The constraints specify that the page has a single column of width CW, with left and right gutters of width 0.6 cm, and that Figures A and B are aligned on the left gutter, and that the column has to be wide enough to fit the figure in. This design is valid for $12.2 \leq PW \leq 26$. An example layout using these constraints is shown in Figure 2.

To accommodate such situations, our model allows the designer to provide multiple *constraint style sheets*. Each style sheet includes constraints that define the layout of the design and that dictate when the design is appropriate. Constraints can be required or annotated with a strength such as "weak" indicating that they are preferred. During manipulation by the viewer, the viewing tool will choose the appropriate style sheet and lay out the document subject to the constraints in the sheet. As the viewer changes the requirements, the document will be redisplayed using the current style sheet if possible. However, if the required constraints become inconsistent with the viewer's desires, the viewing tool will choose another style sheet for the document that is consistent with the viewer's constraints.

For instance, if the viewer of our example document originally displays the document in a window of width 28 cm, then resizes the window to 20 cm, the

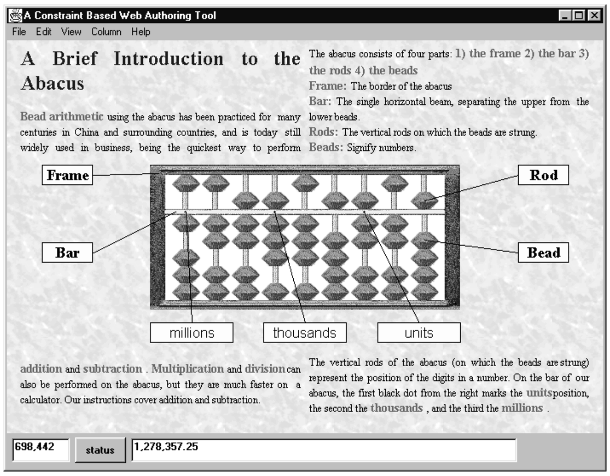

Figure 3: Using Constraints to Position Labels – Two Column Abacus Layout

design will change from two column to one column. If the viewer now resizes it back to 28 cm, the design will change back to two column.

Note that these requirements arising from the browser may be viewed as required constraints. For instance, the constraint

$$PW = 20\text{cm}$$

results when the browser window size is changed. The browser may also place constraints on the values that font sizes can take, for instance

$$TextFontSize \in \{6, 8, 10, 12, 18, 36, 72\}.$$

Constraints may be provided by the viewer, as well as by the designer and the browser software. For instance, a sight-impaired viewer might add the required constraints

$$\begin{aligned} &TextFontSize \geq 16\text{pt} \\ &CaptionFontSize \geq 16\text{pt} \\ &HeadingFontSize \geq 16\text{pt} \end{aligned}$$

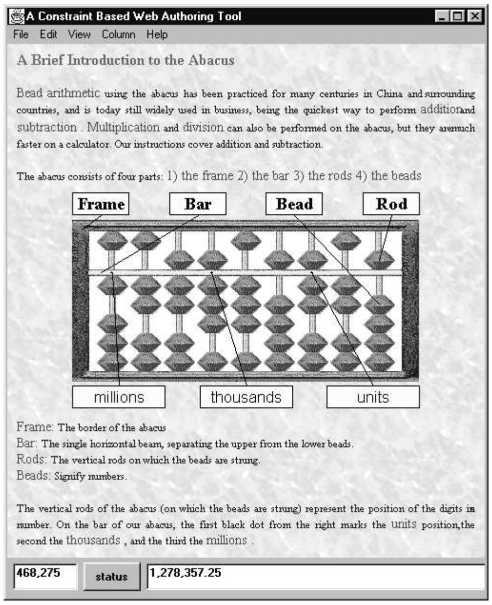

Figure 4: Using Constraints to Position Labels – One Column Abacus Layout

As a more complex example of constraint-based page layout, consider the web page shown in Figures 3 and 4. Figure 3 illustrates a constraint style sheet for a wide page with two column layout, while Figure 4 illustrates a different style sheet for a narrower page with one column layout. In each layout, constraints ensure that the central abacus figure is centered and that the surrounding labels remain appropriately aligned as the window is resized or other edits performed. Invisible "expanding" boxes on either side of the figure ensure that text only appears above and below the figure. Each style sheet contains approximately 200 constraints.

As a final example consider the table layout shown in Figures 5 and 6. Linear arithmetic constraints provide great freedom in laying out elements in a table.

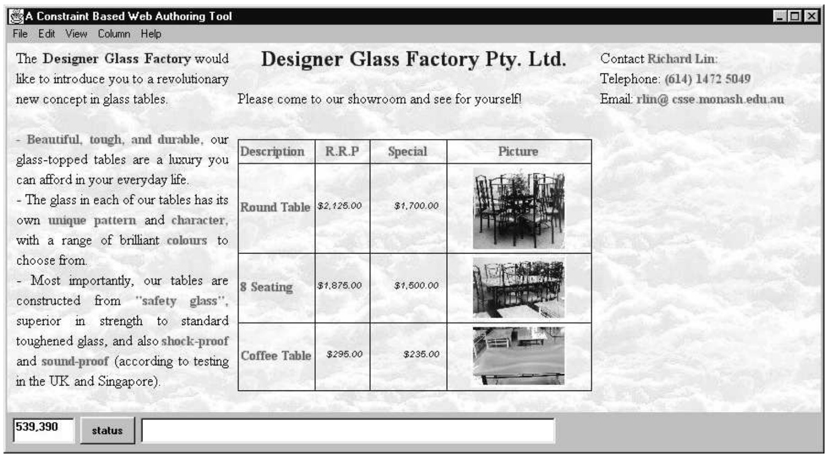

Figure 5: Glass Factory Page - Wide Version

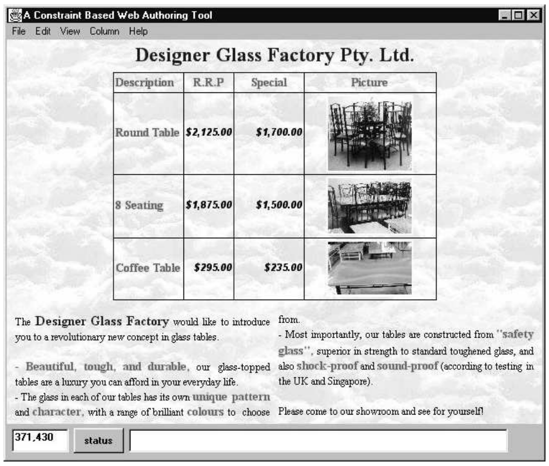

Figure 6: Glass Factory Page - Narrow Version

3 Architecture and Implementation Issues

In this section, we explore possible architectures for a system that supports our constraint-based document layout model, and the requirements that these place on the constraint solver. The constraint-based document layout model has three main components: the *document authoring tool*, the *viewing tool*, and the *constraint solver*.

The authoring tool is used by the designer to construct the constraint style sheets and document contents. Ideally the designer should not need to think in terms of arithmetic constraints or even be aware of the real nature of the constraints. To the designer, they are implicit in various templates and tools such as the "horizontal alignment" tool provided by the authoring tool.

It is natural to require that the authoring tool allows constraint-based editing of the document. In particular, the authoring tool should allow the designer to manipulate the document in exactly the same way as the viewer does, by resizing, changing font size, and so forth. If the designer is unhappy with the design for this choice of values, then the designer should be able to construct an alternative constraint style sheet. Note that this issue of constraint-based editing is orthogonal to the use of constraints in the style sheet — it makes perfectly good sense, for example, to have constraint-based editors for CSS 2.0.

The viewing tool should integrate constraints from the designer with those of the viewer, check which design is appropriate, resolve the constraints, and then display the document contents using the values from the solution to place elements in the layout.

Clearly, the constraint solver is a key component of this architecture. Authors use the constraint solver while laying out and testing the pages while viewers use the constraint system when viewing the page. The demands on the constraint solver, however, are different for authors and viewers.

Authors need the full interactive capabilities of the system. They must be able to add or delete constraints from constraint style sheets and directly manipulate elements of document.

The needs of viewers are much more easily met in the simplest case. Here, the kinds of interaction viewers can have with the constraint system are limited to resizing the browser or a frame and adding constraints on font sizes. However, the set of constraints would be fixed; all that the viewers would change would be constants in the constraints (e.g. browser width). In this case it is possible to pre-compile constraint satisfaction plans using the projection-based algorithm discussed in [10], and just ship compiled Java code to the client and not use a runtime solver. This has the advantages that "constraint solving" on the client will be extremely fast and that the client does not need to download a complex runtime solver from the server.

A more sophisticated constraint-based browser allows the viewer to edit the constraints on the document, for example by adding additional constraints, or by editing the constraints in an applet. This case requires runtime constraint solving by the viewer as well as by the author. It also means that the viewer will need to download the constraint solver the first time it is used. (It can then be stored locally for subsequent reuse.)

In any case it is essential that constraint solving be fast. For example, each time a browser is resized (by either an author or a viewer), the constraints must be resatisfied. This requires that systems of up to several hundred constraints are solved in fractions of a second. For such performance to be possible, the solver needs to recompute a solution incrementally to the same system of constraints. It should also be fast for the designer to add or remove constraints from the design. Again, this requires an incremental approach.

It is also essential that the constraint solver handle underconstrained and overconstrained systems. One situation in which this arises is when moving one component of a web page: we want other components to remain where they are if possible (rather than gyrating wildly and arbitrarily), but at the same time we don't want them to be rigidly locked in place. If a component does need to move, it should move as little as possible. Complex layout problems with conflicting preferences are another cause of overconstrained systems.

As we have seen, one kind of constraints that arise in page layout are linear arithmetic equalities and inequalities. Our experience with other interactive graphical constraint-based systems indicates that simultaneous linear equalities and inequalities arise frequently. In some cases these cyclic collections are inherent in the problem. In others the cycles come about when the author added redundant constraints — a cycle *could* have been avoided by careful analysis. However, this is an added burden on the author. Further, it is clearly contrary to the spirit of the whole enterprise to require web authors — particularly non-programmers — to be constantly on guard to avoid cycles and redundant constraints; after all, one of the goals in providing constraints is to allow users to state what relations they want to hold in a declarative fashion, leaving it to the underlying system to enforce these relations.

Another important class of constraints we have met are finite domain constraints, such as constraints on font sizes, in which each variable has a fixed finite domain of values it can take. In contrast to arithmetic constraints, cycles are less likely to arise with these constraints for web-based applications.

Nonlinear numeric constraints arise less frequently, but are useful for constraining such attributes as angles and areas.

4 Constraint Solving Algorithms

We now briefly describe the two constraint solving algorithms used in the work reported here, namely Cassowary and BAFSS. Both algorithms allow required constraints and preferred constraints whose preferences can be of different strengths. Cassowary handles linear arithmetic equality and inequality constraints. The collection of constraints may include cycles (i.e. simultaneous equalities and inequalities), redundant constraints, and incompatible preferences. BAFSS handles finite domain constraints; however, these constraints are not allowed to contain cycles.

Our current implementation uses BAFSS to solve the constraints relating to font size and Cassowary to solve the page layout constraints. Interaction with each constraint solver can occur in three ways. First, a constraint may be added to the current set of constraints. Second, a constraint may be deleted from the current set of constraints. Finally, the current solution may be manipulated by providing new suggested values for several of the variables.

4.1 An Incremental Simplex Algorithm

Cassowary is an incremental version of the simplex algorithm, specialized for user interface applications. Details of this algorithm are given in reference [6] and we use the Java implementation provided as part of the QOCA constraint solving toolkit [18].

The simplex algorithm is a well-known and heavily studied algorithm for finding a solution to a collection of linear equality and inequality constraints that minimizes the value of a linear expression called the *objective function*. However, commonly available implementations of the simplex algorithm are not really suitable for user interface applications such as the one described in this paper.

The principal issue is incrementality. We need to solve similar problems repeatedly, rather than solving a single problem once, and to achieve interactive response times, very fast incremental algorithms are needed. There are two cases. First, when moving an object with a mouse or other input device, we typically represent this interaction as a one-way constraint relating the mouse position to the desired x and y coordinates of a part of the figure. For this case we must resatisfy the same collection of constraints, differing only in the mouse location, each time the screen is refreshed. This situation arises for both web authors and viewers, since both will be manipulating the web document (or at least resizing the browser). Second, when we first begin a movement, we add a constraint relating the object being moved to the mouse position, and when the movement is completed we remove this constraint. We may also add or remove constraints when interactively editing a layout or applying a new constraint style sheet, and again we would like to make these operations fast by reusing as much of the previous solution as possible. Authors will need this last capability in any event; viewers may or may not, depending on how much flexibility is given to the viewer. The performance requirements are considerably more stringent for resolving a given set of constraints for a new mouse input position than for incrementally adding or deleting a constraint, since in the first case we need to resatisfy the constraints for every screen refresh.

Our constraint solving algorithm is fully incremental, and can be used for both the authoring and viewing tools. (In this case a runtime solver is needed by the viewer's browsing program.) To provide the needed performance, the solver keeps the constraints in a normal form closely related to the basic feasible solved form employed in the simplex algorithm. An incremental version of Gauss-Jordan elimination is used when an equation is added, while an incremental version of the first phase of the simplex is used when an inequality is added.

Constraint deletion is handled by keeping track of how equations and inequalities have been used to create the normal form, that is, which variables they have been used to eliminate. This ensures that removal of a constraint requires only a single pivot.

Non-required constraints are handled by use of a quasi-linear objective function. For instance, imagine that we are editing the variable x and wish to change its value to 50, and the other variables y and z currently have the values 30 and 60. Then the solution we are interested in minimizes the objective function

$$s|x - 50| + w|y - 30| + w|z - 60|$$

where s and w are fixed weights that ensure the strong constraint is always strictly more important than solving any combination of weak constraints. The simplex algorithm cannot directly be used to solve such optimization problems, due to the absolute value operations in the objective function (which make it quasi-linear rather than linear). However, it is possible to transform such problems into an equivalent linear programming problem that can be solved with the second phase of the simplex algorithm.

Resolving of the constraint system for suggested values is done by first identifying which values will be changed (in effect, incrementally adding constraints relating these values to the x and y positions of the mouse or to an input field). The algorithm then produces a data structure that identifies exactly what parts of the normal form need to be updated for changed inputs. In most cases — typically, when the mouse movement doesn't result in one object colliding with another — this update simply involves changing a small number of constants in the normal form. When one object does first collide with another, or moves out of collision, then one or more pivots will usually be required to restore the data structures to their normal form. These pivots are done using a variant of the dual simplex algorithm.

4.2 Solving Acyclic Finite Domain Constraints

In our web application, finite domain constraints are used for font selection. A *hierarchical finite domain problem* consists of n variables $v_1, \ldots, v_n$ such that:

- Each variable v_i has a *finite domain*, $dom(v_i)$, which is the set of values that this variable can take.
- There is a conjunction C of arithmetic *required constraints* over the variables.
- There are *preferred constraints* $D_1, \ldots, D_m$ over the variables.

We associate with each preferred constraint D_i an *error function* e_i that returns the error associated with a particular assignment to the variables. The error function allows us to take into account the strength of the preferred constraints. The error is required to be a non-negative real number and to be 0 if the constraint is satisfied. For instance the error associated with an equation $s = t$ might be $|s - t|$, and for a more important inequality $s \leq t$ it might be $1000|s - t|$ when $s > t$ and 0 otherwise.

A variable assignment θ *satisfies* the problem if $\theta(v_i) \in dom(v_i)$ for each i, and θ satisfies the required constraints C. A *solution* to the problem is a variable assignment θ that satisfies the problem and that minimizes the error $\sum_{i=1}^{m} \theta(e_i)$.

Finding a solution to an arbitrary hierarchical finite domain problem is NP-hard. However, as we have already indicated, the hierarchical finite domain problems arising in document layout have two characteristics which mean that expensive general purpose constraint solving algorithms are not required.

The first characteristic is that the constraints are either *unary* or *binary*, that is to say, at most two variables occur in each required and desired constraint. The second characteristic is that if the system of constraints is viewed as a *constraint graph*, with variables as nodes and an edge between two nodes if there is a required or desired constraint involving those two variables, then this graph is acyclic.

We can use the *BAFSS* algorithm [17] to solve such *binary acyclic hierarchical finite domain problems* in polynomial time. The algorithm uses a straightforward dynamic programming approach. The essential idea is to view the constraint graph as a tree, arbitrarily choosing the root, and then visit the nodes in the tree bottom-up. When a node in the tree corresponding to variable v is visited, for each value in the domain of v the algorithm computes the best partial assignment to the variables whose nodes are lower in the tree.

5 A Prototype Implementation

We have built a prototype of the constraint-based design model. It consists of the three components discussed in Section 3: document authoring tool, constraint solver, and viewing tool. All are written in Java. It should be emphasized that the current implementation is a prototype and not intended for production use by web-page designers. However it is fully functional and demonstrates the technical feasibility of our approach.

The document authoring tool allows the designer to edit the content of document, which consists of text, invisible boxes, images, figures, and tables. Apart from text, each component of the document has a bounding box, and constraints between these boxes dictate the document layout. The document's text flows around these boxes.

A snapshot of the authoring tool is shown in Figure 7. The tool has four windows. The main window shows the document viewed using the current constraint style sheet. The current constraint style sheet is displayed as a textual collection of constraints in two windows: one for the finite domain constraints

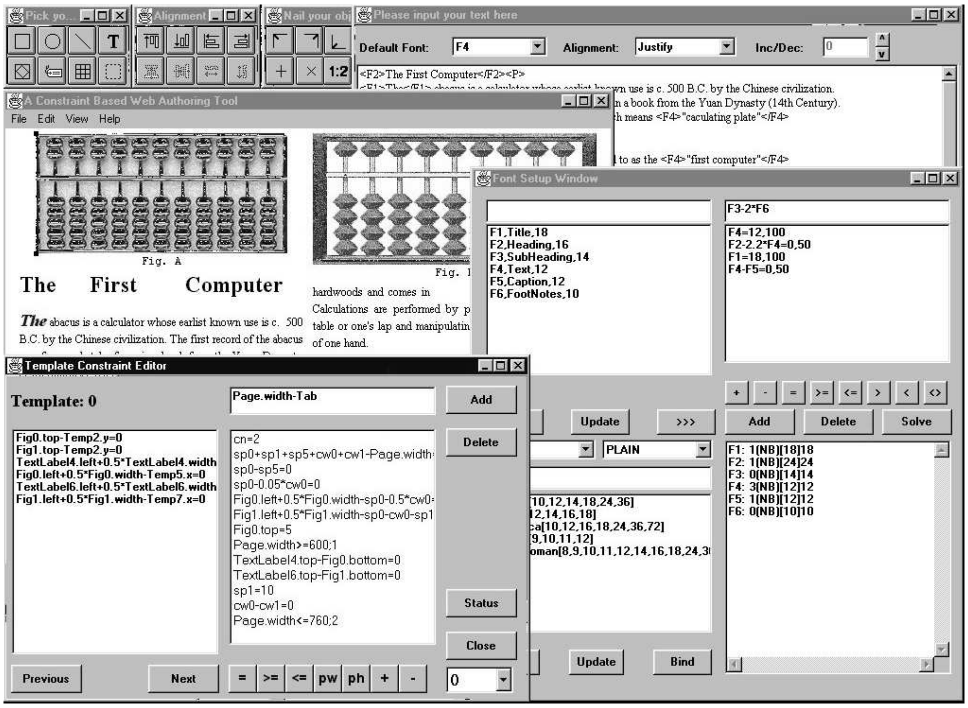

Figure 7: Authoring Tool

controlling the text size and types, and another for the linear arithmetic constraints. The remaining window contains an editor for the document text. Apart from containing the text proper this also contains HTML-like tags indicating paragraphs, line breaks, and different fonts.

The designer can add or delete objects from the document in the main window and use direct manipulation to move objects around. The designer can also directly edit the textual form of the finite domain and linear constraints. To help this process, when adding a new constraint the designer can click on objects in the main window, which will generate the appropriate variable name and attribute, such as `Fig0.left`, in the constraint. The format of the constraints is somewhat more restrictive than that given in our examples: all measurements are in pixels, the righthand side of a constraint must be a constant and integers, e.g. 50 or 100, are used rather than symbolic labels *strong*, *medium*, and *weak* to denote the strength of a preferred constraint.

As an alternative to directly modifying the textual constraints, the designer can use design templates selected in the main window. Currently, there are design templates for one column, two column, and three column layout, for aligning objects vertically and horizontally, for centering objects in a column or page, and for table construction. These add the appropriate textual constraints to the stylesheet. At all times the constraint solver ensures that placement of objects in the document satisfies the current constraints.

The linear arithmetic constraints and finite domain constraints can share variables. When solving the constraints, first the BAFSS algorithm is used to solve the finite domain constraints, then the Cassowary algorithm is used to find values for the remaining variables. This captures the heuristic that font sizes should be chosen before determining document layout.

The designer can manipulate the document in exactly the same way as the viewer does. If at some point the designer is unhappy with the style sheet for the choice of values, then he or she can create an alternate style sheet for that situation. This new style sheet will only contain constraints that cause the document elements to be placed within the document.

To indicate how to move between style sheets, the designer can annotate required constraints within a particular style sheet with a reference to another style sheet for that page. If such a constraint becomes unsatisfiable during interaction with the reader, perhaps after resizing, the other style sheet is tried.

We need to know when a precondition for a constraint style sheet is violated, so that we can determine the new style sheet to jump to. Making the preconditions be required constraints would not address this need, since the solver would only know that a conjunction of ordinary required constraints and required constraints representing preconditions was unsatisfiable. Instead, we make the preconditions be strongly preferred constraints (stronger than any ordinary preferred constraints). Thus, these constraints will always be satisfied unless the original required constraints plus the preconditions are unsatisfiable. During interaction, each of these constraints is checked to confirm that the current solution still satisfies it. If it is not satisfied, then the authoring tool jumps to the associated style sheet.

Currently the viewing tool is derived from the document authoring tool by simply turning off some of the options and only displaying the main window. When a document is first downloaded from a server, the viewing tool and constraint solver are also downloaded. Using parameters from the viewing window, the viewing tool selects a style sheet, uses the constraint solver to solve the constraints and then displays the document contents according to that style sheet. Layout is performed by first determining the placement of the figures and images, and then placing the text around them.

For fast switching between different constraint style sheets, the authoring and viewing tools keep separate instances of the constraint solver(s) for each style sheet. Of course this has a space overhead but makes switching between style sheets very fast.

As noted previously, this is a prototype system, and not one intended for production use. Before being ready for such use, there are some significant questions regarding the user interface to be addressed. This issue is discussed further in Section 8 (Future Work).

6 Evaluation

The performance of our prototype is very encouraging. The authoring tool provides direct manipulation of document elements with style sheets involving several hundred constraints in real time. This is not very surprising and accords with our recent results for user interface construction using a C++ implementation of Cassowary [18].

A more interesting question is the performance of the prototype viewing tool. To give some feel for its behaviour we have measured the performance of our system on five benchmark documents. The first document, *Abacus*, is essentially the sample document from Figures 1 and 2, although we have also added another constraint style sheet for which a sample layout is shown in Figure 8. The second

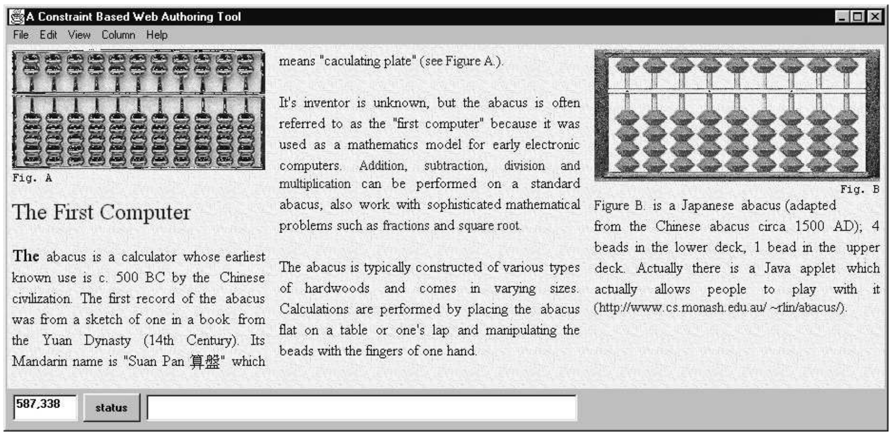

Figure 8: Additional Layout for the Abacus Document

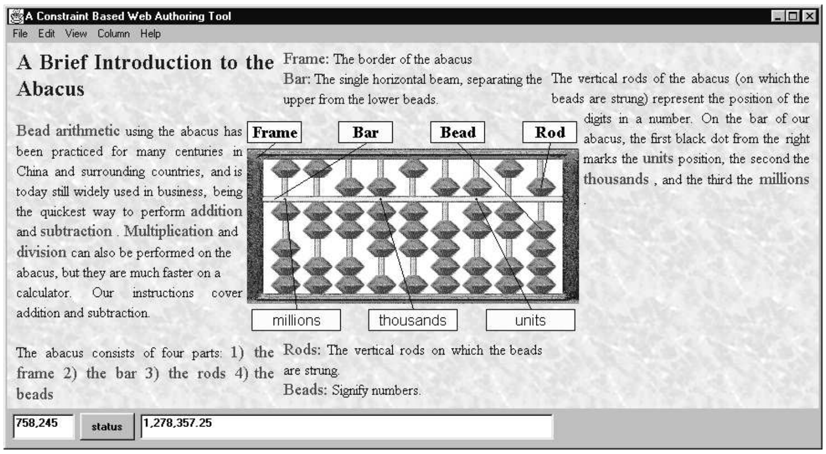

Figure 9: Additional Layout for the Abacus Document with Label Positioning

document, *Labelled Abacus*, is essentially the example shown in Figures 3 and 4, although again we have added another constraint style sheet for which a sample layout is shown in Figure 9. The third document, *Glass Factory*, is essentially the Glass Factory document shown in Figures 5 and 6. The fourth document, *Glass Factory Complex*, is a more complex version of *Glass Factory*. A sample layout for each of the two style sheets is shown in Figures 10 and 11. The final benchmark, *Benchmark Page*, is a web page describing our other benchmarks

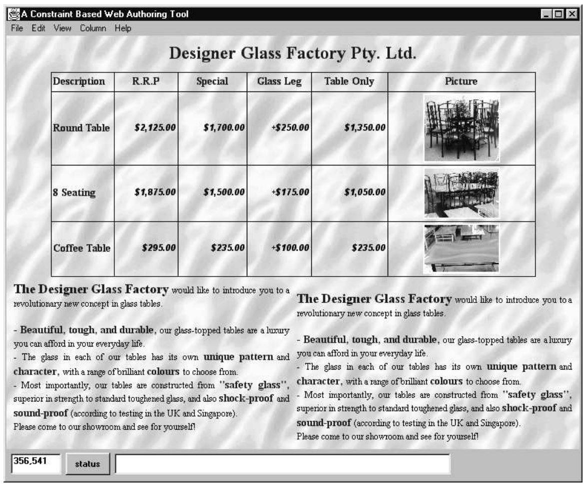

Figure 10: More Complex Glass Factory Page - Wide Version

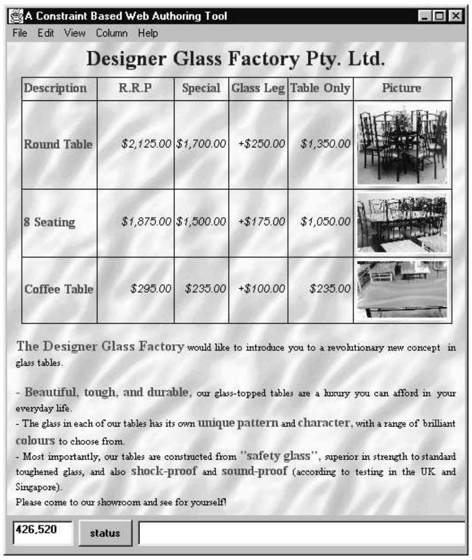

Figure 11: More Complex Glass Factory Page - Narrow Version

— the sort of page that would appear in a web-based version of this paper. Note that the embedded figures of the benchmarks are bitmaps, not the benchmarks themselves. Sample layouts for the two style sheets are shown in Figures 12

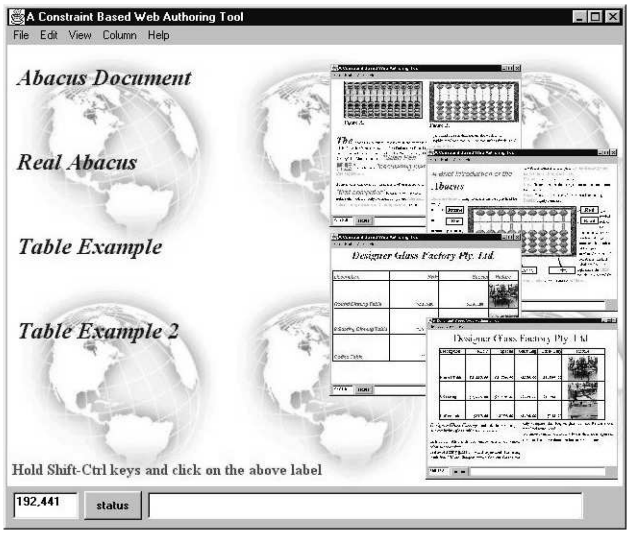

Figure 12: Benchmark Page - Wide Version

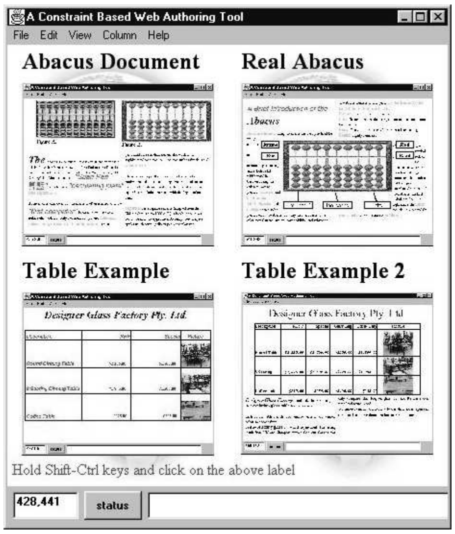

Figure 13: Benchmark Page - Narrow Version

and 13.

In Table 1 we provide some statistics about the benchmarks. First we give

Benchmark	Size (K Bytes)		Style Sheet 1 (# constraint)		Style Sheet 2 (# constraint)		Style Sheet 3 (# constraint)	
	Total	Image	FD	Linear	FD	Linear	FD	Linear
Abacus	57.9	52.9	4	66	3	60	2	66
Labeled abacus	42.7	32.7	3	198	1	191	3	196
Glass factory	25.3	20.5	5	225	4	227		
Glass factory complex	26.4	21.1	5	301	5	298		
Benchmark page	70.8	67.6	1	102	1	100		

Table 1: Information about Benchmarks

Benchmark	Download	Draw	Build Style Sheet		
			# 1	# 2	# 3
Abacus	1200	50	110	160	160
Labeled abacus	1100	110	600	610	650
Glass factory	1150	50	770	1040	
Glass factory complex	1270	50	810	1210	
Benchmark page	500	50	170	160	

Table 2: Evaluation of Browsing Tool

the size of the document in Kbytes. The document consists of the contents plus the style sheets. In addition to the total size of the document, we also give the size of just the images in the document (which account for nearly all of the size). Next we give the number of finite domain and linear arithmetic constraints for each of the two or three style sheets associated with each document.

The viewer also needs a copy of the viewing tool applet. The total size is 493K of which the linear arithmetic constraint solver is 168K, the finite domain solver is 98K and the remaining 217K is the viewer proper. There is considerable redundancy in the linear arithmetic and finite domain solver code so this could be reduced. Of course the viewing tool applet only needs to be downloaded once, and never if it is provided as part of the browser library.

We have evaluated the performance of the viewing tool from a number of viewpoints. Our results are shown in Table 2 All times are in milliseconds running Internet Explorer 4.01 on a Pentium 200MMX with 64M memory. First we give the time to download both the viewing tool applet and the document. This also includes the time for parsing the document. We can see that even if the viewing tool applet must be downloaded, downloading is reasonably fast. Next, we give the average time to draw and redraw the document, without constraint solving. Finally, for each style sheet we give the time taken to initialize the constraint solver for the style sheet. We have also investigated the time taken to switch between stylesheets when the browser is resized and to resolve the constraints when the browser is resized but the current style sheet remains applicable. On all examples for all style sheets the average time is less than one millisecond. Switching between stylesheets is fast because separate constraint solvers are kept for each style sheet (assuming that the constraint solver has already been initialized).

Clearly the time for downloading, switching between style sheets and re-solving the constraints during browser resizing is very reasonable. The time to initialize the constraint solver for each style sheet is also reasonable, taking less time than the download.

7 Related Work

Regarding the web and HTML, constraint style sheets are closely related to Cascading Style Sheets (CSS 1.0 and 2.0) introduced as part of the HTML 4.0 standard [16]. Cascading Style Sheets allow both the author and reader to provide rules that specify various attributes of a web document. Rules can be given weights, which are used to resolve conflicts among rules from different style sheets. A class mechanism provides inheritance of specifications.

The fundamental difference between Cascading Style Sheets and constraint style sheets is that Cascading Style Sheets allow one to specify a particular value for a given attribute (or in some cases as a percentage of another attribute), while constraint style sheets allow general constraints, i.e. partial specifications, to be given for these attributes. For example, a Cascading Style Sheet can include a rule specifying that the left margin of a layout element be a particular value. On the other hand, a constraint style sheet can include an arbitrary linear constraint on the left margin, which might constrain it to be less than twice some other value. (Constraining it to have a particular value is just a special case of the general constraint mechanism.) A similar example is that constraint style sheets allow the designer to specify that separate tables in the document have the same column widths, regardless of the elements in the table, while this is virtually impossible using Cascading Style Sheets since the tables are laid out in isolation from each other. Another difference is that we allow the designer to provide several alternative style sheets for a given document, so that the appearance can change interactively as the viewer resizes the page or chooses different text fonts (though CSS 2.0 does allow media-dependent importation of stylesheets). In addition, we also support figure layout (although reference [16] notes that such an extension to Cascading Style Sheets is expected). On the other hand, Cascading Style Sheets include a class and inheritance mechanism and style rules which aren't provided in our current design.

The `<table>` environment introduced in HTML 3.0 can be viewed as providing certain constraints, including preferences as well as requirements, for example, desired cell width expressed either as an absolute quantity (in pixels) or as a percentage of the total table width. Again, however, there is no general constraint capability.

Regarding constraints, there is a long history of using constraints in user interfaces and interactive systems, beginning with Ivan Sutherland's pioneering Sketchpad system [24]. Most of the current systems use one-way constraints (e.g. [21]), or local propagation algorithms for acyclic collections of multi-way constraints (e.g. [23, 26]). UI systems that handle simultaneous linear equations include DETAIL [12] and Ultraviolet [4]. However, QOCA and the Cassowary Algorithm are the only UI systems we know of that handle both simultaneous linear equations and inequalities [6, 11, 18].

IDEAL [25] is an early system specifically designed for page layout applications. Harada, Witkin, and Baraff [9] describe the use of physically-based modelling for a variety of interactive modelling tasks, including page layout. Their system allows mixed continuous/discrete models, in which a given set of constraints is used until an object being dragged is blocked by geometric constraints; at this point a local search is performed for a nearby state in which all constraints are again satisfied. The U-term language [7] is a more recent constraint-based visual language for specifying the display of data. The DOODLE Visualization Tool [1] provides visualizations of information in an object-oriented database, using U-terms to specify selection and presentation criteria. The interface is implemented in Java, making it accessible from Java-capable web browsers. Another Java-based system is subArctic [13], which provides a one-way constraint solver as part of its Java toolkit (more sophisticated solvers are planned).

Weitzman and Wittenburg [27, 28] have investigated the use of relational grammars for document design. Their work is closely related to ours, since in effect they use a grammar that specifies a class of constraint layout styles. However, their interest is in specifying and recognizing layout styles rather than constraint solving. They only consider rather weak constraint solving techniques based on local propagation. Indeed, it seems rather natural to combine their work with ours.

The work reported here differs from this prior work on constraints and relational grammars in two respects: first, through its support of a negotiation model in which author and reader both contribute constraints that determine the appearance of the document; and second, in the integration of constraints with web documents.

8 Future Work and Conclusion

The principal contributions presented in this paper are:

- A description of how constraints provide important functionality for web page layout and a constraint-based negotiation model for determining the web page appearance.

- An analysis of the requirements on the constraint solver, and of different ways the constraint solving responsibilities can be partitioned between client and server.

- An implemented prototype that demonstrates the feasibility of the idea. The prototype has good interactive performance for both author and viewer, and also demonstrates the feasibility of both server-side and client-side constraint solving.

Our plans for future work include the following projects.

First, we want to support a wider range of constraints for web authors and viewers to use. These will include non-linear arithmetic constraints, and also finite domain constraints for font attributes other than size. We will also investigate a less ad hoc coupling of the BAFSS and Cassowary algorithms.

Second, we want to investigate richer constraint-based web markup languages that incorporate the full features of HTML and CSS along with constraints, and that are insofar as possible extensions of HTML and CSS, so that such languages have a realistic potential for adoption by the web community. Our preliminary work in this area is described in [2].

Third, we plan to design and evaluate better user interfaces for the document authoring tool. We require graphical tools in which the underlying constraints and constraint solver are unobtrusive to the author, but that retain the full power of the constraint-based approach. An important issue is how the designer specifies the layout constraints and how these are visualized. We plan to investigate several different approaches to this question. One approach is to let the designer draw an example page layout first, and then infer constraints from this example [15, 19], or have the designer annotate this page by drawing visual layout constraints, such as frames around text sections to specify alignment—or both. Another approach is "template-based," and requires the designer to select a particular template, such as a centered figure or a two column table, from a predefined palette of layout templates. The template can either be filled with individual page elements (e.g., paragraphs) after its creation or already existing unrelated page elements can be grouped with a newly applied template. A related issue concerns debugging. If a page doesn't appear right for a particular browser configuration, how can we aid the designer in pinpointing the constraints and other information that should be modified? Existing work on constraint debugging [14, 22] provides some starting points, but we need to express the problem to the user graphically and at a higher level, in terms of the web page layout domain. One promising technique is to use animation to show how the page reformats as parameters, such as browser width, vary. Fast incremental constraint satisfaction algorithms, such as those described in Section 4, will be essential here.

Fourth, another potentially important application of constraints is to specify the behaviour of applets used in web pages. Providing applets for animations, simulations, and other kinds of interactive information is an exciting prospect. However, currently such applets are usually produced by writing Java code. This is a time-consuming process. A number of researchers have used constraints for producing simulations and animations without hand-coding the program. For instance, systems such as ThingLab [3] have used constraints for constructing interactive simulations, while systems that support constraint-based animation include Animus [8] and Amulet [20]. These results are all applicable to generating applets. In fact the page shown in Figures 3 and 4 include such an applet for the Japanese abacus. The constraints require each bead to remain on its respective rod, to not pass through another bead, and to remain within the boundaries established by the bars of the abacus. The Java constraint satisfaction code for these applets was produced automatically from the list of constraints using our projection-based algorithm for constraint compilation [10].

In conclusion, we have described an initial foray into applying constraint technology to the web. The results so far are preliminary but very encouraging, and we believe this will be a promising area for future research and subsequent real-world application.

Acknowledgments

We gratefully acknowledge the help of Andrew Kelly and Sitt Sen Chok, who programmed Java versions of the Cassowary algorithm.

This project has been funded in part by the U.S. National Science Foundation under Grants IRI-9302249 and CCR-9402551 and in part by a grant from the Australian Research Council. Alan Borning's visit to Monash University and the University of Melbourne was sponsored in part by a Fulbright award.

References

[1] Michael Averbuch, Isabel, Cruz, Wendy Lucas, and Melissa Radzyminski. As you like it: Tailorable information visualization. Technical report, Database Visualization Research Group, Tufts University, 1996.

[2] Greg J. Badros, Alan Borning, Kim Marriott, and Peter Stuckey. Constraint cascading style sheets for the web. Technical Report 99-05-01, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, May 1999.

[3] A. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[4] A. Borning and B. Freeman-Benson. The OTI constraint solver: A constraint library for constructing interactive graphical user interfaces. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 624–628, Cassis, France, September 1995.

[5] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.

[6] A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 10th ACM Symposium on User Interface Software and Technology*, pages 87–96, 1997.

[7] Isabel Cruz. Expressing constraints for data display specification: A visual approach. In Vijay Saraswat and Pascal Van Hentenryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 443–468. MIT Press, 1995.

[8] Robert Duisberg. Animation using temporal constraints: An overview of the Animus system. *Human-Computer Interaction*, 3(3):275–308, 1987.

[9] Mikako Harada, Andrew Witkin, and David Baraff. Interactive physically-based manipulation of discrete/continuous models. In *SIGGRAPH '95 Conference Proceedings*, pages 199–208, Los Angeles, August 1995. ACM.

[10] Warwick Harvey, Peter Stuckey, and Alan Borning. Compiling constraint solving using projection. In *Third International Conference on Principles and Practice of Constraint Programming*, page To appear, October 1997. Also vailable from http://www.cs.washington.edu/research/constraints.

[11] R. Helm, T. Huynh, K. Marriott, and J. Vlissides. An object-oriented architecture for constraint-based graphical editing. In C. Laffra, E. Blake, V. de Mey, and X. Pintado, editors, *Object-Oriented Programming for Graphics*, pages 217–238. Springer-Verlag, 1995.

[12] H. Hosobe, S. Matsuoka, and A. Yonezawa. Generalized local propagation: A framework for solving constraint hierarchies. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 237–251. Springer-Verlag LLNCS 1118, 1996.

[13] Scott E. Hudson and Ian Smith. SubArctic UI toolkit user's manual. Technical report, College of Computing, Georgia Institute of Technology, 1996.

[14] Walid T. Keirouz, Glenn A. Kramer, and Jahir Pabon. Exploiting constraint dependency information for debugging and explanation. In Vijay

Saraswat and Pascal Van Hentenryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 183–196. MIT Press, 1995.

[15] David Kurlander and Steven Feiner. Inferring constraints from multiple snapshots. *ACM Transactions on Graphics*, 12(4):277–304, October 1993.

[16] H.W. Lie and B. Bos. *Cascading Style Sheets*. Addison-Wesley, 1997.

[17] Richard Lin, Kim Marriott, and Peter Stuckey. Flexible font-size specification in Web documents. In *Proceedings of the Twenty-Second Australasian Computer Science Conf.*, page To appear., Auckland, 1999. Springer-Verlag.

[18] K. Marriott, S.S. Chok, and A. Finlay. A tableau based constraint solving toolkit for interactive graphical applications. In *International Conference on Principles and Practice of Constraint Programming (CP98)*, pages 340–354, 1998.

[19] Brad Myers. *Creating User Interfaces by Demonstration*. PhD thesis, University of Toronto, 1987.

[20] Brad Myers, Robert Miller, Rich McDaniel, and Alan Ferrency. Easily adding animations to interfaces using constraints. In *Proceedings of the 1996 ACM Symposium on User Interface Software and Technology*, pages 119–128, Seattle, November 1996.

[21] Brad A. Myers. The Amulet user interface development environment. In *CHI'96 Conference Companion: Human Factors in Computing Systems*, Vancouver, B.C., April 1996. ACM SIGCHI.

[22] Michael Sannella. Analyzing and debugging hierarchies of multi-way local propagation constraints. In *Proceedings of the 1994 Workshop on Principles and Practice of Constraint Programming*. Springer-Verlag LLNCS 874, 1994.

[23] Michael Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Software—Practice and Experience*, 23(5):529–566, May 1993.

[24] I. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, pages 329–346. IFIPS, 1963.

[25] Christopher J. van Wyk. A high-level language for specifying pictures. *ACM Transactions on Graphics*, 1(2):163–182, April 1982.

[26] Brad Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way dataflow constraints. *ACM Transactions on Programming Languages and Systems*, 18(1):30–72, January 1996.

[27] L. Weitzman and K. Wittenburg. Relational grammars for interactive design. In *IEEE Symposium on Visual Languages*, pages 4–11, 1993.

[28] L. Weitzman and K. Wittenburg. Automatic presentation of multimedia documents using relational grammars. In *ACM Multimedia Conference*, pages 443–451, 1994.