A Constraint Extension to Scalable Vector Graphics

Greg J. Badros[1] Jojada J. Tirtowidjojo[2] Kim Marriott[2] Bernd Meyer[2]

Will Portnoy[3] Alan Borning[3]

[1] InfoSpace, Inc., 2801 Alaskan Way, Suite 200 Seattle, WA 98121, USA
greg.badros@infospace.com

[2] School of Computer Science and Software Engineering, Monash University, Clayton, Victoria 3168, Australia
{jojada, marriott, berndm}@mail.csse.monash.edu.au

[3] Dept. of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350, USA
{will,borning}@cs.washington.edu

Abstract

Scalable Vector Graphics (SVG) is a language that describes two-dimensional vector graphics for storage and distribution on the Web. Unlike raster image formats, SVG-based images scale nicely to arbitrary resolutions and sizes. However, the current SVG standard provides little flexibility for taking into account varying viewing conditions, such as different screen formats, and there is little support for interactive exploration of a diagram. We introduce an extension to SVG called Constraint Scalable Vector Graphics (CSVG) that permits a more flexible description of figures. With CSVG, an image can contain objects whose positions and other properties are specified in relation to other objects using constraints, rather than being specified in absolute terms. For example, a box can be specified to remain inside another box, without being given an absolute position. The precise layout can then be left to the browser, which can adapt it dynamically to changing viewing conditions on the client side. Further extensions add support for alternate layouts, interaction, and declarative animation. Leveraging well-established methods for linear constraint solving, we implemented a prototype viewer for CSVG by embedding our Cassowary constraint solver into an existing SVG renderer.

Keywords: constraints, differential scaling, semantic zooming, interaction, CSVG, SVG, Scalable Vector Graphics.

1. INTRODUCTION

The Scalable Vector Graphics (SVG) language [16] is a language based on XML [11] developed by the World Wide Web Consortium for describing two dimensional vector graphics for storage and distribution on the Web. SVG can dramatically improve graphics on the Web. In contrast to raster image formats such as GIF, JPEG, and PNG, which store a matrix of individual pixels that compose an image, an SVG image contains instructions for resolution independent rendering, so that the same SVG file can be shown in more detail when viewed at a higher resolution. SVG files are also more compact than raster images, easier to process and analyze, integrate well with Cascading Style Sheets (CSS) [10] and can make full use of the Document Object Model (DOM) [2]. Given these advantages, SVG is increasingly well supported by both commercial and free software.

1.1 SVG is not enough

Although the SVG format is a huge step forward for many kinds of images, we can do even better for diagrammatic illustrations. In particular, SVG does not provide for flexible layout given different viewer requirements and browser capabilities, such as screen format and font preferences. Moreover, the support for interaction and animation in SVG is very limited. In this paper we demonstrate how extending SVG with constraints provides the basis for improving these aspects, thus significantly enhancing the flexibility of SVG. This is a modular extension to SVG that ensures upward compatibility with the original format.

A constraint is a declarative specification of a relationship that we wish to hold true. For example, "`Format` appears to the left of `DateFormat`" is a constraint. We can write the constraint mathematically as: $\text{Format}.x_{\text{right}} + \text{horiz\_spacing} \leq \text{DateFormat}.x_{\text{left}}$. In this constraint, we only specify the properties we wish to hold, but do not give concrete values, or an explicit method (an algorithm) for making the properties hold. The task of finding appropriate values is delegated to a constraint solver that is used during rendering.

For example, assume that we want to display a part of the Java object hierarchy (Figure 1), but we want different layouts to be used in different viewing conditions so as to convey the information as clearly as possible. In a landscape format, we may want to use a diagram such as Figure 2, whereas a portrait format would be better served by the layout of Figure 3. Although these diagrams look very different, they have the same logical (topological) structure and semantics. Using SVG alone for this purpose presents a problem, because we are required to give absolute positions and sizes for the graph-

WWW10, May 1-5, 2001, Hong Kong.
ACM 1-58113-348-0/01/0005.

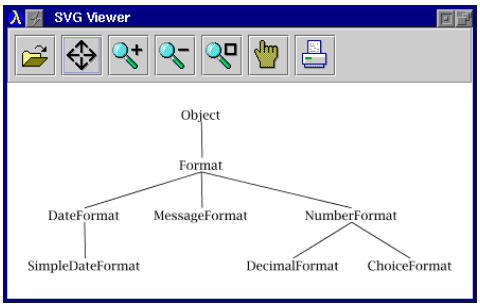

Figure 1: SVG image diagramming the object hierarchy for the `Java.Text.Format` class. The SVG source for this image appears in Figure 5.

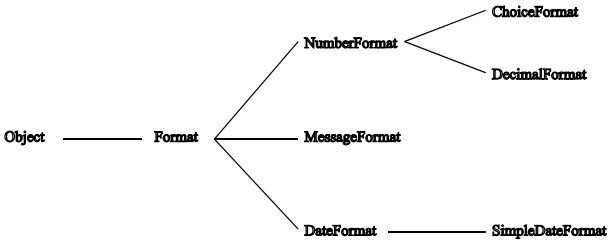

Figure 2: CSVG rendering of the `Format` class hierarchy inside a wide and short viewport.

ics objects. The only flexibility that SVG allows is uniform scaling of the entire graphic.

Another limitation of SVG is that it provides little direct support for interactive exploration of a diagram. In particular, the static nature of SVG forces authors to write scripts if they wish to maintain the layout relationships of objects in the diagram automatically, for example, when an object in the diagram is manipulated interactively. This task is tedious, complicated, and tightly coupled to the specific illustration. The new layout ideally should be generated automatically by the browser based on a more abstract layout specification.

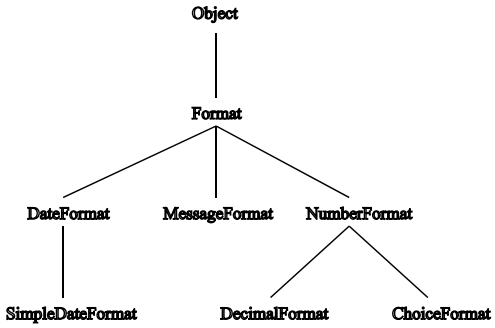

Figure 3: CSVG rendering of the `Format` class hierarchy inside a narrow and tall viewport.

1.2 Interactive manipulation & animation

As another example, consider the Local Area Network (LAN) topology diagrams shown in Figure 4. The network, shown in Figure 4(a), connects three different buildings — A, B, and C — via a site-wide backbone. Shrinking the whole diagram uniformly may cause buildings A, B, and C to become too small to display their internal details. By using an alternative presentation for each of the buildings we can support a more powerful form of resizing. Figure 4(b) shows a possible result. Zooming that preserves the semantic presentation of the LAN topology diagram but changes its appearance is called *semantic zooming.*

Semantic zooming in combination with user interaction can allow viewers to explore a diagram interactively, selecting subcomponents they wish to see in detail and hiding the details of other sub-components. Of course the diagram layout must change dynamically, preserving connections and also effectively utilizing the available screen space.

We would also like SVG to provide a form of zooming that we term *differential scaling.* (Both semantic zooming and differential scaling are kinds of focus+context techniques [31].) In contrast to uniform scaling, differential scaling enables us to enlarge a part of a diagram to see more detail, while simultaneously shrinking another part, so that the overall screen area for the diagram is kept constant. This capability clearly involves scaling various parts of the figure differently. An important application of this non-uniform scaling is to the sizing of text in diagrams. A sight-impaired user, for instance, may want to shrink the overall size but to increase text font size in labels. As an example, see Figure 4(c), which contains the same graphical components as Figure 4(a) but with an increased text size for labels and smaller boxes. Note that this means different line breaking for the text in the labels.

We would also like viewers to be able to change the layout of objects in the diagram interactively, without changing its semantic content or logical structure. Users should thus be restricted to performing *semantics preserving manipulations.* For example, Figure 4(d) shows a resized and rearranged diagram of the local area network example. Here, the system preserves the connection information when the user extends the width of building C's bounding rectangle and interactively moves the LAN segment boxes as well as the repeater circle.

The idea of interactive manipulation and exploration of a diagram naturally generalizes to an interaction metaphor in which the viewer uses the diagram as a simulation of some physical artifact and can explore "what-if" scenarios by direct manipulation and other interaction, as in the original ThingLab system [6]. In section 3.2, we discuss an interactive abacus.

The same principle can be further generalized to provide constraint-based animation [14]. In addition to making the properties of some graphical objects dependent on the properties of other objects, and allowing the user to manipulate these properties interactively, we can also make these properties dependent on a timer variable. Using a constraint that relates object properties to a time offset gives us an immediate way to specify trajectories and other animation properties. In section 3.3, we discuss a seesaw animation.

1.3 Extending SVG

The above examples demonstrate how semantic zooming, differential scaling, semantics preserving manipulation, and

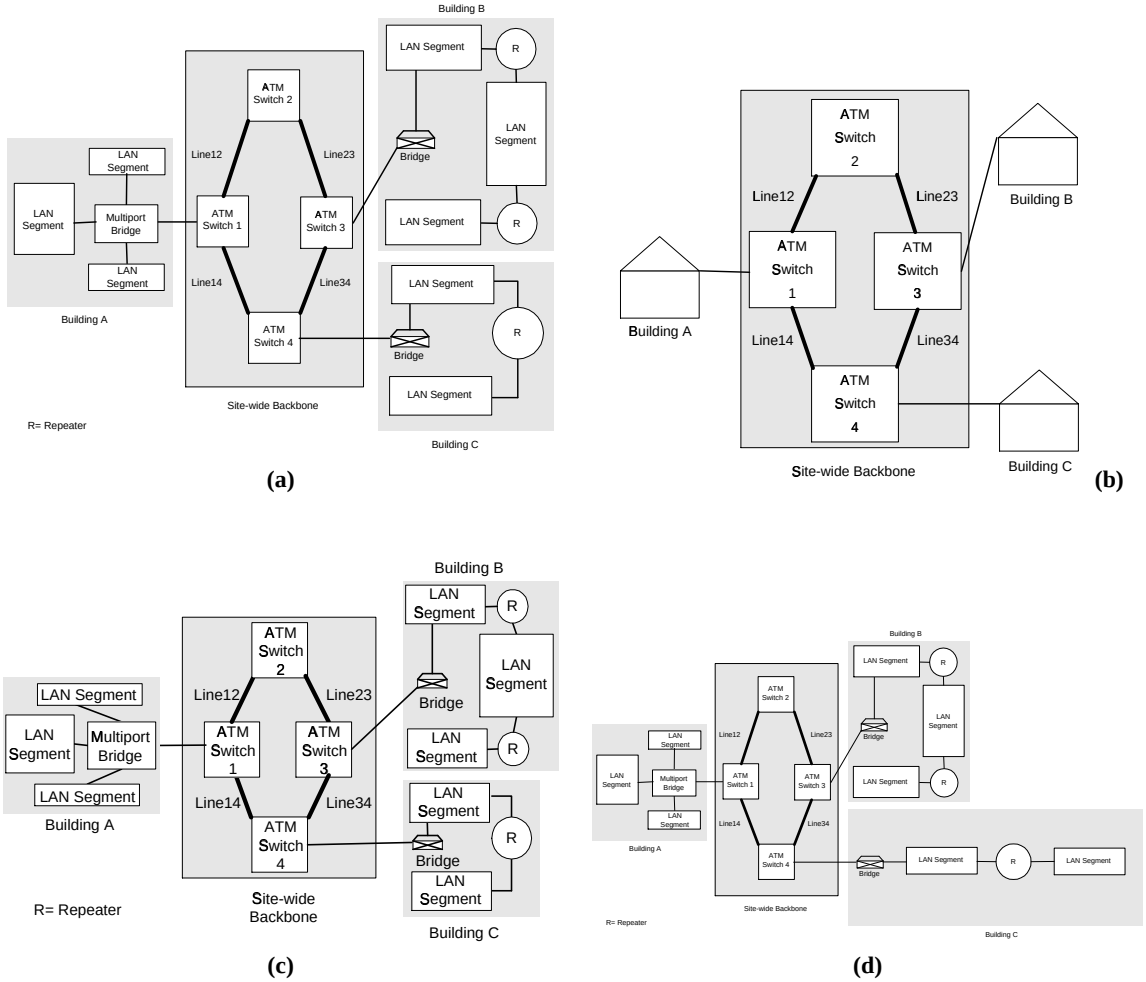

Figure 4: Various diagrams of a local area network. Relative to diagram (a), (b) illustrates semantic zooning, (c) demonstrates differential scaling, and (d) is an example of the output of a semantics-preserving transformation.

animation greatly improve SVG's value. We shall show that constraint-based layout specification of SVG elements can be used as the basis for such dynamic layout and provide SVG with the desired capabilities. These constraint-based layout specifications are more abstract and flexible specifications of layout. In our Constraint Scalable Vector Graphics (CSVG) extension to Scalable Vector Graphics, images can use arbitrary linear arithmetic constraints to control the layout of shapes, lines, paths, and font sizes.

The other primary additional capability provided by CSVG is allowing alternate layouts for the same logical group of components in a diagram. Which layout is chosen depends on testing preconditions at runtime, such as display window size or aspect ratio. The layout may also be chosen interactively by the user, allowing exploration of the diagram.

Our main technical contributions are:

- a motivation for using constraints and alternative layouts for a wide class of SVG diagrams;
- a description of Constraint Scalable Vector Graphics as an extension of SVG; and
- a prototype implementation of a CSVG viewer based on the CSIRO SVG viewer [32]. The prototype makes use of the sophisticated constraint solving algorithm Cassowary [9].

2. SVG BACKGROUND

In spirit, the Scalable Vector Graphics format is very similar to the PostScript page description language [1], but it uses XML syntax instead of postfix notation. Figure 5 gives an example.

In SVG, each element describes a shape to be rendered. For example:

```
<rect x="20" y="10"
      width="10" height="5"/>
```

describes a rectangle whose top-left corner is positioned at coordinate (20,10) with a width of 10 units and a height of 5 units. Lengths and coordinates can include explicit units, but when they are omitted, the user space coordinate system is used [16, Ch.7].

An especially powerful SVG element is `path`. Its `d` (for "data") attribute contains a string that encodes a command-based description of an arbitrary outline. For example, the element:

```
<path d="M 20 10 L 30 10 L
         30 15 L 20 15 Z"/>
```

describes a rectangle path equivalent to the preceding `rect` element: first **M**ove to (20, 10), then draw **L**ines to (30,10), (30,15), and (20,15), and finally close the path (**Z**). Uppercase command characters designate the use of absolute coordinates, while lowercase denotes relative coordinates. Other `path` sub-language commands include **C**urve-to, **S**mooth curve-to, **Q**uadratic Bezier curve-to, and more.

Other important elements include `defs` and `use` for defining objects and later referencing them, `image` for embedding raster image files (e.g., PNG or JPEG graphics), `text` for including text, and `g` for grouping sub-elements to be rendered as a single entity.

```
<?xml version="1.0"?>
<!DOCTYPE svg SYSTEM "svg.dtd">
<svg
  width="4.5in" height="4in"
  viewBox="0 0 100 100"
  style="fill: none; font-size: 15;
         stroke-width: 1; stroke: black;
         text-anchor: middle">
 <desc>The object hierarchy surrounding
       the class "Java.text.Format"</desc>

 <text x="200" y="30">Object</text>
 <text x="200" y="90">Format</text>
 <text x="60" y="150">DateFormat</text>
 <text x="60" y="210">SimpleDateFormat</text>
 <text x="200" y="150">MessageFormat</text>
 <text x="380" y="150">NumberFormat</text>
 <text x="310" y="210">DecimalFormat</text>
 <text x="450" y="210">ChoiceFormat</text>
 <line x1="200" y1="32" x2="201" y2="75"/>
 <line x1="200" y1="92" x2="60" y2="135"/>
 <line x1="200" y1="92" x2="201" y2="135"/>
 <line x1="200" y1="92" x2="380" y2="135"/>
 <line x1="60" y1="152" x2="61" y2="195"/>
 <line x1="380" y1="152" x2="310" y2="195"/>
 <line x1="380" y1="152" x2="450" y2="195"/>
</svg>
```

Figure 5: SVG source of the class hierarchy illustration shown in Figure 1.

The `g` element (as well as the basic shape elements) has an attribute `transform` whose value is a list of translating, scaling, rotating, or skewing transformations that will be applied to the grouped shape before final rendering. One of the more complex features of SVG is that objects may be positioned and dimensioned *relative* to a transformation frame or in *absolute* units.

SVG also contains several animation elements that describe time-based perturbation of the containing object. These elements can be used to achieve motion along paths, the fading in or out of objects, changes in color, and more. For example, to animate moving a rectangle horizontally across the viewport to the right, we write:

```
<rect x="20" y="10"
      width="10" height="5"/>
  <animate attributeName="x"
           attributeType="XML"
           begin="0s" dur="9s"
           fill="freeze"
           from="20" to="120"/>
</rect>
```

A program that reads an SVG file has access to the internals of the image via the SVG Document Object Model [16, Appendix B]. The DOM [2] permits direct access to the SVG element tree in an object-oriented manner and allows the ma-

nipulation of element structures and attributes. For example, to increase the size of a text element, we can execute the following code in ECMAScript [15], a standardized version of JavaScript:

```
e = document.getElementById("TextElement");
e.setAttribute("transform", "scale(2)");
```

and the selected element will be scaled to twice its normal size. The SVG DOM can be used in combination with scripting and event handlers (e.g., `mousedown`, `onclick`) to permit some useful interactive capabilities.

3. CSVG: CONSTRAINT SCALABLE VECTOR GRAPHICS

The conventional means of delivering an image across the Internet is in a rasterized image format such as PNG or JPEG (Figure 6). The resolution is fixed when that file is created, and the artifact the user receives is inflexible. The adoption of the SVG image format permits a different delivery mechanism (Figure 7). The high-level image description is stored in the SVG image format, preserving much of the semantic value provided by the author. That SVG file is then sent across the network. An SVG renderer on the client side chooses the resolution and creates a rasterized display of that image specially tuned for the display device and the desired size.

The key observation concerning the evolution from raster images to SVG is that we are sending a higher-level description across the network and moving some of the processing of the image from the server side to the client side. Thus, the artifact sent across the Internet is more flexible—it can be used as the source for generating a high-quality printout of the image, to create a low-resolution thumbnail of the image, or even to "render" the image aurally using speech synthesis to describe the diagram. The decision of how to present the image is made with input from the user, and from the browser and other client-side software. Style sheets provide yet another way to increase the flexibility of the image sent over the network: not only is the resolution left undetermined, but the final decision as to such aspects as the coloring scheme can be delayed until after applying style sheet declarations.

Our constraint extension to SVG permits describing the author's layout intentions, and defers the actual positioning and sizing of the image's elements until just before final display for the user (Figure 8). To support this greater flexibility, we have made four extensions to the SVG language.

First, we store all SVG standard element attributes in predefined constraint variables and support identifier names in addition to literal numbers. These identifiers may refer to other constraint variables in the document. For example, the `rect` SVG element:

```
<rect id="rectA" x="10" y="20"
      width="square_edge"
      height="square_edge"
      rx="5" ry="5"/>
```

will have six predefined variables based on its `id` attribute: `rectA_x`, `rectA_y`, `rectA_width`, `rectA_height`, `rectA_rx`, and `rectA_ry`; one for each attribute value. Because identifiers are found in its attributes, this `rectA` will have both `rectA_width` and `rectA_height` variables constrained to be equal to the `square_edge` variable.

Constraint variables may also be declared explicitly by using a new `var` element, in which case they have as optional attributes a stay weight, `sw`, an edit weight, `ew` and a preferred value, `value`. The stay weight indicates the "inertia" of the variable—how strongly it likes to keep its current value—while the edit weight indicates how important it is to change this variable during interaction when it is directly manipulated. Although numeric values can be used for weights, it is usual to use symbolic names such as strong and weak.

An important use of stay weights is to enforce stability in graphical layout. Every constrained attribute implicitly has a weak stay constraint which states that its value only be changed as little as possible. Such stay constraints make objects remain in place unless some other stronger desire forces a change.

Second, we add another new element type called `constraint`. Each `constraint` element has a required attribute, `rule`, and an optional attribute, `strength` and specifies an equation or inequality that we wish to hold for constraint variables. An example of such a CSVG constraint and its related object tag is:

```
<constraint rule="square_edge >= 50"
            strength="strong"/>
<rect x="10" y="20" width="square_edge"
      height="square_edge"/>
```

which defines a square whose edge length is at least 50.[1] The rule implicitly introduces the new constraint variable, `square_edge`.

Constraint strengths allow the designer to specify a relative ordering of the importance of constraints. If there is a conflict, the stronger constraints will be given preference. There are different ways to resolve conflicts among constraints with the same strength [7]; in our implementation we minimize the weighted sum of their errors.

Given a system of constraints, the constraint solver must find an assignment to the variables that satisfies the required constraints exactly, and the solution should satisfy the preferred constraints as well as possible, giving priority to the more strongly preferred constraints and taking into account edit and stay weights.

Third, we add several built-in read-only constraint variables. (A read-only variable is one that cannot be changed by the solver to satisfy the constraint in which it occurs [7].) Two variables, `viewport_width` and `viewport_height`, are used to allow the image to be influenced by the size of the display area. We expose `current_time` and `current_time_squared` which are both ever-increasing read-only variables that allow CSVG to support the declarative specification of time-based animations more directly than the `animate` elements.

Fourth, we have added alternate layouts for groups of SVG elements. For this purpose a new `docase` element containing

[1] Our syntax was chosen for simplicity and conciseness. We also considered using MathML [23] syntax for constraint specification: it has the advantage that standard XML parsers can parse the constraint properly, but is considerably more verbose and complicated.

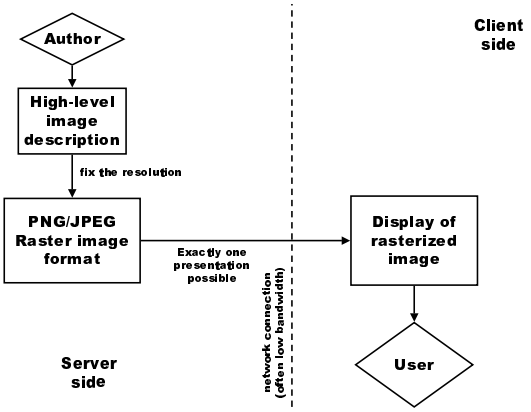

Figure 6: Conventional process of delivering a raster image across the network.

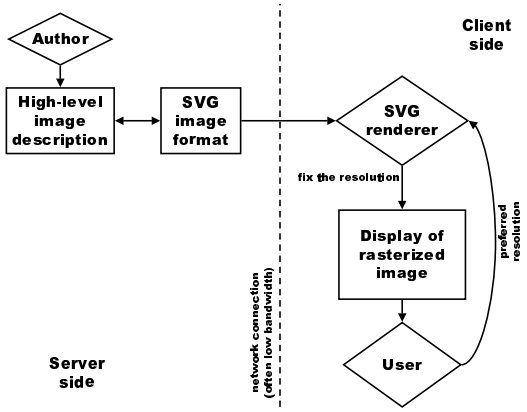

Figure 7: Process of delivering a resolution-independent SVG image across the network.

one or more `gcase` element(s) is defined. Each `gcase` element specifies an alternative version, with the version chosen depending on which pre-condition holds. The `gcase` element simply extends the SVG `g` element with a `case` attribute.

As an example, consider Building C of the LAN diagram in Figure 4. The Bridge rectangle needs to have two different connecting points for its connection line. In the initial condition shown in Figure 4(a), where the Bridge does not completely stay to the left of the LAN segments, its connecting point is on top edge. In other condition such as shown in Figure 4(d) the connecting point is on right edge. Clearly, the Bridge needs alternate layouts. By using a `docase` element, the alternative layouts can be expressed as follows:

```
<!-- Connection to BridgeC -->
<constraint rule="connectionLine_x1 =
                    To_BridgeC_x1"/>
<constraint rule="connectionLine_y1 =
                    To_BridgeC_y1"/>

<docase id="caseBridgeC">
  <var id="To_BridgeC_x1"/>
  <var id="To_BridgeC_y1"/>
  <gcase id="bridgeC_handleOnRight"
         case="leftC1_x1 >= bridgeC_x +
                        bridgeC_width" >
    <constraint rule="To_BridgeC_x1 =
           bridgeC_x + bridgeC_width"/>
    <constraint
      rule="To_BridgeC_y1 = bridgeC_cy"/>
  </gcase>
  <gcase id="bridgeC_handleOnTop"
    case="bridgeC_x + bridgeC_width >=
                              leftC1_x1">
    <constraint
      rule="To_BridgeC_x1 = bridgeC_cx"/>
    <constraint
       rule="To_BridgeC_y1 = bridgeC_y"/>
  </gcase>
</docase>
```

Such group alternatives were used in the LAN example in

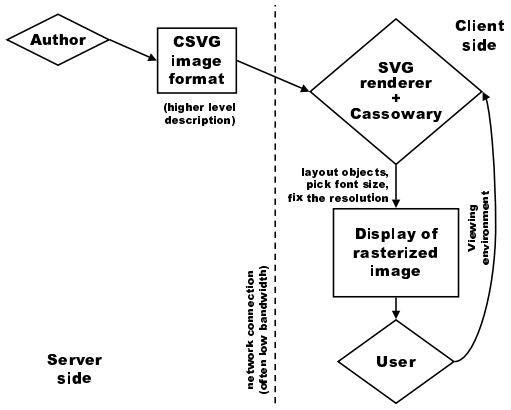

Figure 8: Process of delivering a CSVG image across the network.

Figure 4 to collapse subdiagrams when the available display area becomes too small. Note that such pre-conditions must be re-checked when viewing parameters change, because the truth values can change dynamically (for example, due to resizing). Therefore they, too, must make use of the constraint-solver. However, they treat the constraint variables as read-only.

Pre-conditions are not the only way to choose a group case. It is also possible to switch to an alternative case interactively in the browser. Consider a scenario where the browser collapses some sub-diagram because of a lack of display space, but the user needs to see the details of this group. The user can thus decide to expand the subgroup explicitly, but since the overall size of the diagram will exceed the display area, the user will have to employ scrolling to view the rest of the diagram.

3.1 A layout example

We can rewrite Figure 5 to specify constraints on the layout of the class hierarchy, rather than giving exact locations for

all the parts of the illustration. Our CSVG description of the image looks like the ordinary SVG image (Figure 1) under "ideal" viewing conditions. However, the CSVG file is far more flexible, and it will appear as shown in Figures 2 and 3 when the viewport dimensions are altered. An ordinary SVG file would always appear as just a uniformly scaled version of Figure 1.

For our CSVG version of the class hierarchy, we use a total of 109 constraints. Each viewing condition, either portrait or landscape format, employs 53 constraints that reflect typical layout desires for viewing trees: nodes at the same level are aligned horizontally (8 constraints), different levels are spaced at equal vertical intervals (8 constraints), and all lines must connect at the midpoints of nodes (28 constraints). Of the remaining 9 constraints for each alternative format, 8 are used to maintain the midpoint of nodes and 1 is used to divide the viewport into columns. The other constraints are for defining constants such as vertical interval and the viewport midpoints. An abridged version of the CSVG source appears as Figure 9. See section 4 for timing details of this and following examples.

3.2 An interactive example

Consider the abacus shown in Figure 10, inspired by an earlier constraint-based applet [8]. The diagram allows the user to select and move beads on the abacus. Each bead is constrained to behave like the physical object and so must move along the rod and cannot pass through another bead or through the bar. For example, these constraints ensure that if the user grabs a bead and pushes it upwards all the beads above it will automatically be pushed with it.

This interactive abacus CSVG file has a total of 185 constraints and 419 variables. Each column of beads has 37 constraints that keep the beads attached to the rod (7 constraints), maintain the bead size (7 constraints), ensure the beads to stay in the frame (4 constraints), and keep the beads in their relative order (19 constraints).

3.3 An animation example

Constraints relating object positions to the current time can be used to support simple animations. Layout constraints are even more compelling when parts of the image are moving: the positions of the remaining objects can be described at a high level, knowing that the solver will animate whatever other objects need to move to maintain the specified desires.

Figure 11 shows four screenshots of our CSVG prototype rendering an animation of a ball falling on a seesaw. The `seesaw.csvg` image contains 18 constraints to support the animation: 12 for the positions of the various elements, 1 relating the ball to the `current_time_squared` built-in variable, 1 stating that the ball must remain above the left edge of the seesaw, and 4 describing that the seesaw can go neither through the floor nor through the fulcrum.

4. IMPLEMENTATION

On the client side of the pipeline, we have implemented a CSVG viewer to experiment with the additional expressiveness it provides. Our prototype is based on version 0.71 of the CSIRO SVG Viewer [32] which is implemented in Java and uses IBM's XML4J parser version 2.0.15 [22]. Figure 12 presents the system architecture.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg SYSTEM "svg-19990812.dtd" >
<svg style="fill:none; stroke:black;
           stroke-width:1; font-size:16 ">
 <!-- Defining constants -->
 <var id="topMargin" value="5"
      sw="strong" ew="weak"/>
 <var id="sideMargin" value="5"
      sw="strong" ew="weak"/>
 ...
 <constraint rule="middleVertical =
                     viewport_width / 2"/>
 <constraint rule="verticalSpace =
                     viewport_height / 4"/>
 <constraint rule="middleHoriz =
                     viewport_height / 2"/>
 ...
 <docase id="PortraitOrLandscape">
  <gcase id="PortraitFormat"
         case="viewport_width &lt; 800">
   <!-- Elements in Portrait Format -->
   <text id="object">Object</text>
   <text id="format">Format</text>
   ...
   <line id="object_format"/>
   <line id="format_dtf"/>
   ...
   <!-- Divide viewport width into columns -->
   <constraint rule="columnWidth =
                       viewport_width / 4"/>
   <!-- Maintain the midpoint of nodes -->
   <constraint
      rule="object_cx =
              object_x + object_width / 2"/>
   <constraint
      rule="format_cx =
              format_x + format_width / 2"/>
   ...
   <!-- Aligning nodes horizontally -->
   <constraint
      rule="object_cx = middleVertical"/>
   <constraint
      rule="format_cx = middleVertical"/>
   ...
   <!-- Aligning vertically
       at equal intervals -->
   <constraint
       rule="object_y = topMargin + 15"/>
   <constraint
       rule="format_y =
               object_y + verticalSpace"/>
   ...
   <!-- Attached lines to nodes -->
   <constraint rule="object_format_x1 =
                       middleVertical"/>
   <constraint rule="object_format_x2 =
                       middleVertical"/>
   <constraint rule="object_format_y1 =
                       object_y + textMar-
gin"/>
   <constraint
       rule="object_format_y2 = format_y -
                  format_height - textMar-
gin"/>
   ...
  </gcase>
  <gcase id="LandscapeFormat"
         case="viewport_width >= 800">
    ...
  </gcase>
 </docase>
</svg>
```

Figure 9: CSVG source of the object hierarchy for the `Java.text.Format` class.

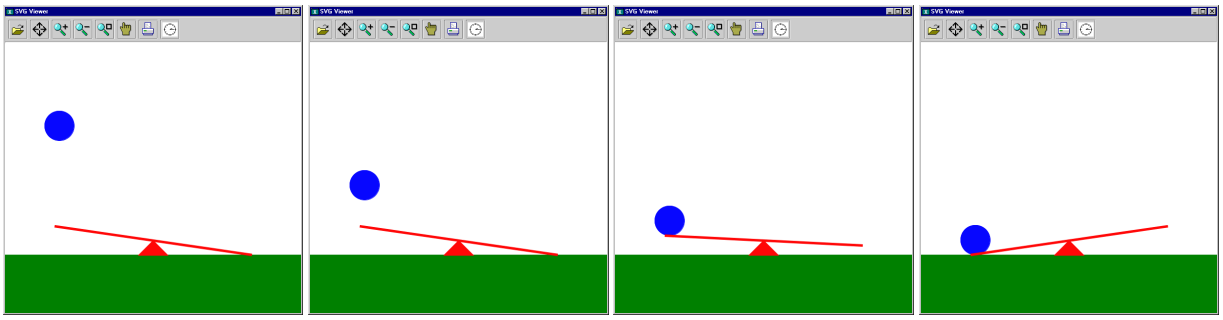

Figure 11: CSVG animation of a ball falling towards seesaw. The position of the ball is directly related to the square of the time offset, and the seesaw moves because of constraints describing its behavior.

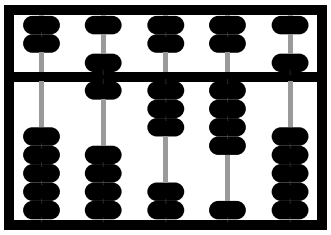

Figure 10: Abacus with Constraint-based Interaction

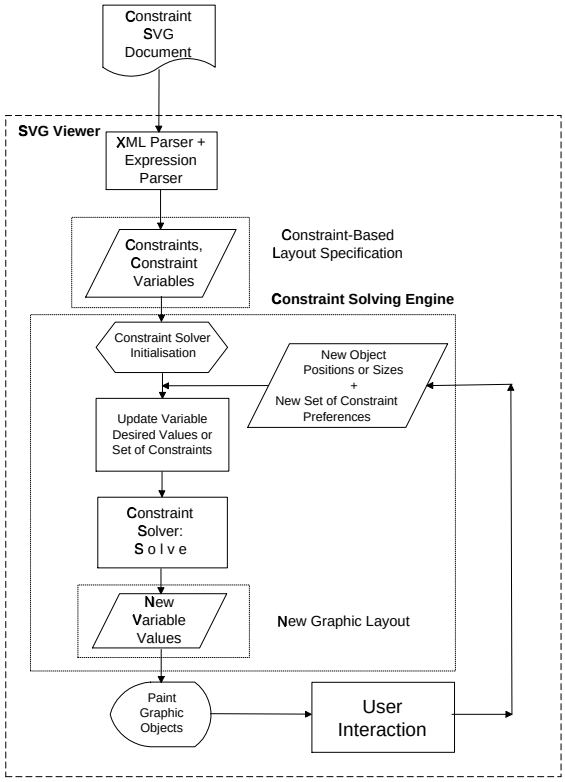

Figure 12: Architecture of our implementation.

As with any XML language, CSVG is defined by its Document Type Definition. Our CSVG DTD is a straightforward extension of the SVG DTD. To the list of permissible children of SVG group elements (`svg` and `g` elements), we added the `var`, `constraint`, and `docase` elements. Then, we added the definition of each new element and its attributes.

After the XML parser reads in the SVG document, we create all the variable and constraint objects of the document, then we feed them to a constraint solving engine. When the solution is available, the viewer renders the graphic objects. Every time a user initiates a resize or movement of an object, the new size or coordinate position of the object is used for setting the associated constraint variables' desired values. The constraint engine then is called to find another solution that reflects the new presentation of all graphic objects in the layout. The same process happens when the validity of group pre-conditions changes due to interactive manipulation. When switching to a different `gcase` element, the constraints and variables of the previous `gcase` element are removed from the engine, and the new set of constraints and variables is added before we re-solve the system.

The constraint solver is a core part of the implementation. The current implementation employs a Java implementation of the Cassowary constraint solving algorithm [3, 9, 27]. This solver supports arbitrary linear arithmetic constraints (both equalities and inequalities over real-valued variables). Cassowary handles cycles without difficulty and handles both required and preferred constraints. The algorithm it employs is an incremental version of the simplex algorithm that we have optimized for interactive graphical applications.

When rendering the figure, we retrieve the values of attributes from their associated constraint variables. For `path` elements, we prefix names of constraint variables with the `$` symbol to avoid ambiguity. For example, we write:

```
<path d="M $x $y l $dx $dy"/>
```

to move to the absolute coordinates held in `x` and `y`, and then draw a line to the relative coordinates contained in variables `dx` and `dy`.

We tested the performance of our prototype using a Dual PentiumII 450 MHz machine running Java 1.2 with the HotSpot virtual machine under Windows NT 4.0. For the class hierarchy example, adding the constraints and the initial solve requires 266 ms. Subsequent re-solves of the constraint system when changing viewing format requires less than 200 ms. For the abacus example, adding all of its constraints and variables to the solver and letting the solver initialize itself requires 485 ms. Subsequent re-solves of the constraint system when moving a bead along its rod require about 16 ms each.

5. RELATED WORK

There is a long history of using constraints in interfaces and interactive systems, beginning with Ivan Sutherland's pioneering Sketchpad system [35]. Juno-2 is a more recent constraint-based drawing application [20]. Constraints have also been used in several other layout applications. IDEAL [37] is an early system specifically designed for page layout applications. Harada, Witkin, and Baraff [18] describe the use of physically-based modeling for a variety of interactive modeling tasks, including page layout. GLIDE [33] 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 [28, 29], and Badros and Stachowiak [5] uses constraints for window layout.

We have previously introduced CCSS, an extension of Cascading Style Sheets with constraints [4]. Our CSVG motivation and philosophy is analogous to that of CCSS, and CCSS is directly applicable to controlling style properties of CSVG documents as well. The primary addition of CSVG beyond CCSS is the ability to control non-style properties of SVG elements. This feature is necessary to control layout because the positions of those objects are determined not by style properties but by element attributes. An earlier paper [8] had goals similar to CCSS, but did not integrate as well with the emerging web standards.

Other style sheet languages including PSL (Proteus Style Language) [26], DSSSL (Document Style Semantics and Specification Language) [24], and XSL (eXtensible Style Language) [12], delay finalizing various presentational attributes of a figure until later in the delivery process, closer to the viewing user. None of these style languages, however, attempt to preserve layout desires to perform layout dynamically on the client side.

Diehl and Keller describe constraint extensions to the Virtual Reality Markup Language (VRML) [13]. Their extension provides both local propagation constraint solving and also finite domain constraint solving. However, it does not provide arbitrary linear arithmetic constraints which we believe are the minimum required to adequately model geometric attributes.

The animation aspects of SVG and CSVG are related to the Synchronized Multimedia Integration Language (SMIL) [21]. Another project called Madeus has used the Cassowary solver to handle a wider range of constraints in multimedia documents [36]. Madeus provides support for both temporal and spatial relationships, and includes a rudimentary authoring environment.

6. CONCLUSIONS & FUTURE WORK

Our constraint extension to SVG provides useful new expressiveness for describing illustration graphics at a higher semantic level. CSVG permits deferring the actual layout of the objects in the figure until client-side rendering, thus resulting in greater flexibility in dealing with varied viewing environments and user desires. In particular, it allows semantic zooming, differential scaling, and semantics preserving manipulation and interaction. In addition, it provides a unifying basis for interaction and animation support.

There are substantial opportunities for future improvements of CSVG. Constraint-based graphics documents are difficult to author, and currently there are no authoring environments for generating CSVG at the appropriate level of abstraction. It is essential that drawing programs permit users to specify constraints interactively, maintain them dynamically throughout editing, and ultimately reflect those constraints in the saved CSVG file. Constraint-based graphics editors such as Cooldraw [17], Aldus Intellidraw™, and Noth's CDA [30], as well as SVG-capable editors such as Adobe Illustrator™ or Sketch [19], may provide a useful starting point.

Even in the presence of graphical editing tools for CSVG, it may be beneficial to provide some syntactic extensions for CSVG. Future versions of CSVG could support referencing other elements' attributes directly. It may also be useful to permit even higher-level constraint abstractions in the CSVG source. For example:

```
<align dir="horizontal" anchor="middle">
  <!-- basic shape objects here -->
</align>
```

would permit easier specification of the intention that a set of basic shapes are aligned in a row by their vertical centers. Constraints at this level also avoid problems that arise when object structure changes. Suppose a basic shape is removed from a diagram (e.g., manipulating the SVG DOM). In such a case, should indirect relationships through that object remain or be removed? If only the primitive constraints are present, the situation is ambiguous. With multiple objects being aligned with a single declaration, the answer is clear—those objects should remain aligned.

An important direction for future work is to extend the power of the constraint solving algorithm, which currently only allows linear constraints. This limitation is not implicit in the design of CSVG, but is imposed by the solvers used in the current implementation. In particular, we are working on support for arbitrary one-way and multi-way constraints [34]. For example, a text element in a CSVG document could be constrained to display the coordinates of a circle: moving the circle would update the string, and editing the string would move the circle.

Another area for future work is to better describe the semantics of SVG in terms of constraints and constraint hierarchy theory. This direction is similar to what we did for Constraint Cascading Style Sheets [4] and it may provide a unifying implementation mechanism for existing aspects of SVG as well. In particular, some of the scripting events, such as `onMouseMove`, may be handled within this framework: a discrete action (such as a button press) establishes a connection that then is managed via a constraint relationship until a subsequent action removes the constraint [25].

In conclusion, CSVG provides a surprising amount of expressiveness at a minimal implementation complexity and a low performance cost.

Acknowledgments

Thanks to Vincent Hardy of the SVG Working Group for his helpful feedback on our work, and to Jeffrey Nichols and Denise Pinnel for their comments on a draft of this paper. Also thanks to Jeffrey for his work on the CSVG web site. This research has been funded in part by a U.S. National Science

Foundation Graduate Research Fellowship and the University of Washington Computer Science and Engineering Wilma Bradley fellowship for Greg Badros, in part by NSF Grant No. IIS-9975990, and in part by an Australian ARC Large Grant.

7. REFERENCES

[1] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1990.

[2] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document object model (DOM) level 1. W3C Recommendation, October 1998. `http://www.w3.org/TR/REC-DOM-Level-1`.

[3] G. Badros and A. Borning. The Cassowary linear arithmetic constraint solving algorithm: Interface and implementation. Technical Report UW-CSE-98-06-04, University of Washington, Seattle, Washington, June 1998. `http://www.cs.washington.edu/research/constraints/cassowary/cassowary-tr.pdf`.

[4] G. Badros, A. Borning, K. Marriott, and P. Stuckey. Constraint cascading style sheets for the web. In *Proceedings of the 1999 ACM Conference on User Interface Software and Technology*, November 1999.

[5] G. Badros and M. Stachowiak. Scwm—The Scheme Constraints Window Manager. Web page, 1997-2000. `http://scwm.sourceforge.net/`.

[6] 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.

[7] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992. `http://www.cs.washington.edu/research/constraints/theory/hierarchies-92.html`.

[8] A. Borning, R. Lin, and K. Marriott. Constraints for the web. In *Proceedings of 1997 ACM Multimedia Conference*, 1997.

[9] A. Borning, K. Marriott, P. Stuckey, and Yi Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 1997 ACM Symposium on User Interface Software and Technology*, October 1997.

[10] B. Bos, H. Lie, C. Lilley, and I. Jacobs. Cascading style sheets, level 2. W3C Working Draft, January 1998. `http://www.w3.org/TR/WD-css2/`.

[11] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C Recommendation, February 1998. `http://www.w3.org/TR/REC-xml`.

[12] J. Clark. XSL transformations. W3C Recommendation, November 1999. `http://www.w3.org/TR/xslt`.

[13] S. Diehl and J. Keller. VRML with constraints. In *Proceedings of the Web3D-VRML 2000 fifth symposium on Virtual reality modeling language*, Monterey, California, February 2000. `http://www.cs.uni-sb.de/RW/users/diehl/VRMLCONSTR/VRMLConstr.html`.

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

[15] ECMAScript language specification, 3rd ed., December 1999. `ftp://ftp.ecma.ch/ecma-st/Ecma-262.pdf`.

[16] J. Ferraiolo. Scalable vector graphics (SVG) 1.0 specification. W3C Working Draft, December 1999. `http://www.w3.org/TR/1999/WD-SVG-19991203/`.

[17] B. Freeman-Benson. Converting an existing user interface to use constraints. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 207–215, Atlanta, Georgia, November 1993.

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

[19] B. Herzon. Sketch, a vector drawing program for unix. Web page, 2000. `http://sketch.sourceforge.net/`.

[20] A. Heydon and G. Nelson. The Juno-2 constraint-based drawing editor. Technical Report 131a, Digital Systems Research Center, Palo Alto, California, December 1994.

[21] P. Hoschka. Synchronized multimedia integration language. W3C Recommendation, June 1998. `http://www.w3.org/TR/REC-smil/`.

[22] IBM AlphaWorks. XML for Java. `http://www.alphaworks.ibm.com/tech/xml4j`.

[23] P. Ion and R. Miner. MathML. W3C Recommendation, July 1999. `http://www.w3.org/TR/REC-MathML`.

[24] ISO/IEC. Document style semantics and specification language (DSSSL). ISO/IEC 10179, 1996.

[25] R. J. Jacob, L. Deligiannidis, and S. Morrison. A software model and specification language for non-wimp user interfaces. *ACM Transactions on Computer-Human Interaction*, 6(1):1–46, March 1999. `http://www.acm.org/pubs/articles/journals/tochi/1999-6-1/p1-jacob/p1-jacob.pdf`.

[26] P. Marden, Jr. and E. Munson. PSL: An alternate approach to style sheet languages for the world wide web. *Journal of Universal Computer Science*, 4(10), 1998. `http://www.cs.uwm.edu/~multimedia`.

[27] K. Marriott, 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.

[28] B. Myers, D. Giuse, R. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet: Comprehensive support for graphical highly interactive user interfaces. *IEEE Computer*, November 1990.

[29] B. Myers, R. McDaniel, R. Miller, A. Ferrency, A. Faulring, B. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, June 1997.

[30] M. Noth. Constraint drawing applet. Web page, 1998. `http://www.cs.washington.edu/research/constraints/cda/info.html`.

[31] R. Rao and S. Card. The table lens: merging graphical and symbolic representations in an interactive focus + context visualization for tabular information. In *CHI'94 Conference Proceedings*, pages 318–322, 1994.

[32] B. Robinson and D. Jackson. SVG toolkit. Web page, 1999–2000. `http://sis.cmis.csiro.au/svg/`.

[33] K. Ryall, J. Marks, and S. Shieber. An interactive constraint-based system for drawing graphs. In *Proceedings of the 1997 ACM Conference on User Interface Software and Technology*, Banff, Alberta Canada, October 1997.

[34] M. 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.

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

[36] L. Tardif, F. Bes, and C. Roisin. Constraints for multimedia documents. In *Proceedings of the Second International Conference and Exhibition on the Practical Application of Constraint Technology and Logic Programming*, Manchester, United Kingdom, April 2000.

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