Authoring Adaptive Diagrams

Cameron McCormack, Kim Marriott and Bernd Meyer
Clayton School of Information Technology, Monash University
Clayton, Victoria, Australia
{Cameron.McCormack, Kim.Marriott, Bernd.Meyer}@infotech.monash.edu.au

ABSTRACT

The web and digital media requires intelligent, adaptive documents whose appearance and content adapts to the viewing context and which support user interaction. While previous research has focussed on textual and multimedia content, this is also true for diagrammatic content. We have designed and implemented an authoring tool which supports the construction of adaptive diagrams. Adaptive layout behaviour is specified by using constraint-based placement tools as well as by allowing the author to specify more radical layout changes using alternate layout configurations. As well as specifying alternate layouts, the author can specify alternate representations for an object, alternate styles and alternate textual content. The resulting space of different versions of the diagram is the cross product of these different alternatives. At display time the version is constructed dynamically, taking into account the author specified preference order on the alternatives, current viewing environment, and user interaction.

Categories and Subject Descriptors

I.3.6 [**Computer Graphics**]: Methodology and Techniques—*Interaction techniques*; I.7.2 [**Document and Text Processing**]: Document Preparation—*Format and notation, Photocomposition/typesetting*

General Terms

Algorithms, Design

Keywords

diagrams, adaptive layout, authoring

1. INTRODUCTION

There is widespread recognition that the web and digital media requires intelligent, adaptive documents whose appearance and content adapts to the viewing context and which support user interaction [5, 7, 16]. Previous research has focussed on textual and multimedia content and largely ignored adaptation of diagrammatic content, treating this as a kind of image. This is unfortunate, since charts, maps, plans, networks and other diagrammatic notations are an important, commonly used vehicle for communication and can equally benefit from adaptive presentation.

Scalable Vector Graphics (SVG) [3], the web graphics format, provides some support for adaptive presentation of diagrams. It supports zooming and uniform rescaling of diagrams, media-specific styling of graphical content through CSS style sheets [2, 11], and allows alternate versions of document elements for different media and languages. While support for more sophisticated adaptation and interactive behaviour currently requires scripting, there have been a number of proposals to extend SVG with more sophisticated layout capabilities [12, 14, 15].

However, there has been virtually no research into how to author adaptive diagrams. It is not reasonable to expect web developers to have to directly write SVG and script to encode standard adaptive behaviour. Diagram authoring tools are required that hide this encoding away from all but the most expert web developer. The main contribution of this paper is the design of the first adaptive diagram authoring tool. Such a tool should be easy to use and, ideally, it should naturally extend the authoring model used in existing diagramming tools. It should also be general enough to allow the author to specify the most useful kinds of adaptive behaviour for a wide variety of different kinds of diagrams. It is not at all clear how to reconcile these two aims and this provides the major difficulty in the authoring tool design.

In order to determine the kinds of adaptive behaviour that the tool should support, we examined diagrams from a wide variety of application areas and detailed how each type of diagram could be sensibly adapted to different viewing environments. We identified seven main kinds of adaptation–change of layout, style, form, textual content, and focus and the introduction/removal of indirection and animation. These are described more fully in Section 2. Figures 1 and 2 illustrate the kinds of adaptation that we believe are required.

In Section 3 we describe an authoring tool for constructing diagrams that exhibit the kinds of adaptive behaviour we identified. Layout changes can be divided into relatively minor layout adjustment, in which the layout changes smoothly in response to changes in text or minor changes in the viewport dimensions, and major structural rearrangement of the layout. Thus, our tool allows the author to specify minor layout changes using constraint-based placement tools and also allows the author to specify structurally different layout alternatives called *configurations*. A similar approach has been previously used for authoring adaptive textual documents [1, 7] and adaptive multimedia documents [8, 9, 17].

One of the main innovations in the design is a consequence of the observation that object representation, style changes and text changes are largely orthogonal to layout changes. Thus, the tool allows the author to specify alternate representations for an object, alternate layout configurations, alternate styles and alternate text. The resulting space of different versions is the combination of all these different alternatives. At display time the version is constructed dynamically by solving the associated constraints, taking into account the author specified preference order on the alternatives, the current viewing environment, and previous user interaction.

A key aspect of the tool design is the choice of constraint-solving algorithms. These need to be powerful enough to support the kinds of high-level constraints that naturally arise in diagrams, such as text boxes, alignment and distribution. On the other hand we did not want to require the browser to provide sophisticated constraint solving abilities, rather we wanted to be able to compile the diagram and its adaptive behaviour into a currently supported graphics standard, namely a combination of SVG and script. This contrasts with previous constraint-based adaptive document authoring tools which have required the same constraint solving capabilities in the browser and in the authoring tool. To allow this have chosen to use multi-way propagation based constraint solving methods, originally developed for interactive graphical applications in the authoring tool since they can be readily compiled into one-way constraints, which in turn are readily implemented in script. This is possible because in the final document there are typically only a few, pre-determined ways of interacting with it and so the flow of information is fixed.

While the authoring tool is still a prototype, we believe it demonstrates that it is possible to build a diagram authoring tool that is reasonably easy to use yet allows specification of powerful adaptive behaviour. Furthermore, the resulting diagrams and adaptive behaviour can be readily compiled into SVG with scripting. The three main contributions of the current paper are: (1) a systematic identification of the reasons for adapting a diagram and the different kinds of adaptation; (2) a description of a model for authoring adaptive diagrams that naturally extends the authoring model of current diagramming tools while allowing specification of powerful adaptive behaviour; and (3) investigation of the constraint-solving capabilities required by both the browser and authoring tool.

2. KINDS OF DIAGRAM ADAPTATION

A necessary first step in designing our tool for authoring adaptive diagrams was to identify the kinds of adaptive behaviour that diagrams can usefully exhibit. We examined nearly 200 diagrams from a wide range of application areas and from a variety of academic journals, newspapers, text books and the web. The examples were chosen to cover the most common types of diagrams (such as charts, trees and networks, flow diagrams, maps and processes). For each example we thought about how the presentation of the diagram could be adapted and the reasons for doing so.

	Layout	Style	Focus	Form	Indirection	Text	Animation
Display	✓		✓	✓	✓	✓	
Medium		✓					✓
Accessibility		✓		✓			
Internationalisation	✓	✓				✓	
Variable content	✓					✓	

Table 1: The main kinds of adaptation in diagrams and the primary reasons for using them

Our study found five main reasons for modifying or adapting the presentation of a diagram. Since these are similar to those for adapting the presentation of other sorts of content [10], we do not go into details. They are:

- **Display size and resolution:** The dimension and resolution of presentation spaces varies, requiring the layout to be modified to make best use of the available space.
- **Medium capabilities:** A diagram should adapt to the presentation medium and viewing device capabilities.
- **User accessibility and preferences:** The diagram may need to be modified in order to make its content more accessible to particular users or to cater for their style preferences.
- **Internationalization:** Another example of responding to user requirements is adapting to different languages and cultural conventions.
- **Variable content:** Some adaptation may be required if (parts of) the diagram are only generated at display time.

Other, less important reasons for adaption we identified were contextual information about other elements on the page the diagram is displayed with, information about the user history or interests, and the need to cater for low bandwidth.

We identified seven distinct kinds of adaptation which can be usefully applied to real-world diagrams. Table 1 summarises the main uses of these different kinds of adaptation with respect to the reasons identified above.

- **Layout:** Layout involves changing, repositioning and resizing diagram elements. The obvious application is to adapt the layout to the available viewing space by, for instance, collapsing whitespace, scaling the diagram, or changing its orientation. Changes in text may also lead to layout changes and adaptation to a language with a different reading order may lead to significant layout changes.
- **Style:** Style captures the non-geometric aspects of the diagram elements, such as colour or choice of font. Style change is required to cater for user accessibility and preferences and to device capabilities. For instance, it is required when adapting a diagram utilising colour to one that is suited to monochrome printing or presentation to colour-blind users. Style change is also important to ensure that the choice of colours and symbols takes account of cultural associations. It can also be used to adapt to the style of the surrounding context.
- **Focus:** Focussing is used to emphasise more important parts of a diagram and to reduce (or hide completely) peripheral information. Example of focussing techniques are user controlled zooming, fish-eye lens and semantic zooming in which the user controls the level of detail shown in the diagram. Focussing supports adaptation to the view-

ing space size, to user interests and can be used to reduce the size of a diagram for faster downloading. Diagrams that have repeated items in them often use some syntax, such as an ellipsis, to show that some of the items have been omitted (where the number of items omitted might depend on the amount of space available).

- **Form:** A more extreme kind of adaptation is to change the way in which the information is represented by the diagram. For instance, hierarchical data can be shown using layered trees or using tree maps. Another example is to change a diagram to a completely textual description of the communicated information. Changes in form can be used to reduce the spatial extent of the diagram, improve accessibility to blind users, and to reduce bandwidth requirements.
- **Indirection:** Indirection refers to graphics conventions that provide information about graphic elements in another part of the diagram and use some sort of correspondence to relate that information back to the appropriate graphic elements. Introducing indirection can reduce the size of a diagram. One example is a legend; rather than have labels next to all elements of a diagram, the elements can be coloured and these colours associated with the labels in the legend. This way, duplicate text strings are eliminated, freeing up space in the diagram. Another example of indirection is moving text labels into a key, away from the elements that they label, and replacing them with numbers that are an index into the key. This is useful, for instance, when preparing a tactile version of a diagram since Braille labels are usually much larger than the non-Braille labels and so may not fit into the diagram.
- **Text:** Clearly internationalisation may require the textual content of a diagram to be adapted. Alternate, equivalent text that is shorter than the original text can be used to conserve space, also.
- **Animation:** Animation can be introduced or removed from a diagram to adapt to the capabilities of the display device. If a diagram is conveying change, for example, the changes might be visualised using animation. For a printed paper version of the diagram, however, the change could be represented by rendering snapshots of the diagram at various points in the animation.

We also need to consider user interaction. This has two broad roles. One is *control* of the the adaptation of the diagram. For example, to control the center of focus or level of detail, the font size or the size of the viewport. The other role is *exploration* of the diagram in order to better understand the information behind the diagram. For example, by highlighting relations (e.g. highlighting the corresponding entry in the legend when moving the mouse over a data point in a chart). Such exploration can be information mutating, such as changing the inputs to a diagram that demonstrates a process or computation.

As an example of an adaptive diagram, consider Figure 1, which is a labelled illustration of the structure of the Earth.[1] The first version of the diagram shows a cutaway of the earth with three text boxes labelling parts of the cutaway. Horizontally, the canvas is divided into two columns with a 65%/35% split. The text boxes are distributed vertically along the right side of the canvas, and are sized to be just tall enough to contain their text for their given widths. The second version of the diagram shows how the text box positions adapt to the available vertical space: when there is insufficient space to position the text boxes without overlap, they are instead flowed vertically from the top of the canvas, the bottom of one box aligning with the top of the next. In addition, the diagram has adapted to the viewer clicking on the second label, thereby collapsing it. Finally, the third version of the diagram shows adaptation due to an even smaller canvas size, this time introducing indirection for the labelling, allowing the viewer to point at hotspots on the diagram to reveal the text. It has also adapted the style to a viewing device that does not support colour, such as an e-book reader.

Figure 2 gives another example. This diagram shows the steps involved in cell division. It has three layouts: a horizontal layout, a vertical layout and an interactive version. All layouts adapt to small changes in the available space by resizing the arrow lengths and also re-wrapping the text labels so that they fit between the cells (or in the case of the interactive version, between the arrows and the canvas boundary). If the horizontal layout is displayed, and the available rendering space on an interactive device shrinks such that there is insufficient space for the text to fit between the cells, the vertical layout will be switched to. Finally, if there is insufficient space for either of these, the interactive version will be shown. In this version, the viewer can click the blue buttons to navigate (change the focus) between the steps. Note also that the interactive version shows the diagram adapted to the preferred language of the viewer.

3. AUTHORING ADAPTIVE DIAGRAMS

While the kinds of adaptive behaviour identified above can be obtained by hand-coding scripted SVG, it is not reasonable to expect web developers to have to do this. Therefore, diagram authoring tools that allow the interactive specification of adaptive behaviour are required, so that authors need not delve into such low-level implementation.

There are a number of conflicting design requirements on any such tool. First, the tool should not be too difficult to learn to use. Second, the adaptive behaviour of a diagram must be understandable and predictable by the author so that diagrams can be authored with a clear idea of how they will respond in different viewing environments. Third, specifying adaptive behaviour should not be too burdensome, or else authors will not do so. Fourth, the resulting adaptive diagrams should be able to be compiled into a reasonably compact representation supported by current standards such as SVG with scripting. And fifth, it should be general enough to support the seven kinds of useful adaptive behaviour identified above for a wide variety of diagrams.

However, building an authoring tool that supports all kinds of diagrams seemed overly ambitious. We decided that we would target diagrams whose layout is "grid-like" in the sense that the objects in the diagram are placed with respect to horizontal and vertical grid-lines. This encompasses a wide variety of diagrams, including the examples in Figures 1 and 2. However, it does not include, say radial layout of trees or organic style layout of networks. We also decided to focus primarily on the first four reasons for adaptation and consider only a limited kind of variable con-

[1]The examples in this paper use colour to demonstrate adaptation to the capabilities of the output medium and thus may be better viewed via the online proceedings.

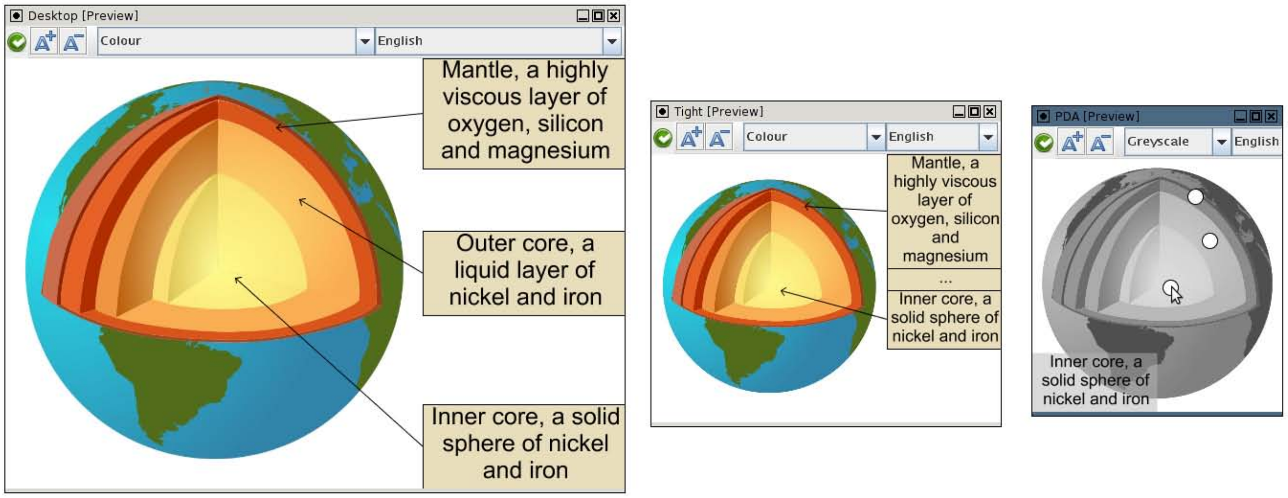
Desktop [Preview]
Colour
English
Mantle, a highly viscous layer of oxygen, silicon and magnesium
Outer core, a liquid layer of nickel and iron
Inner core, a solid sphere of nickel and iron
Tight [Preview]
Colour
English
Mantle, a highly viscous layer of oxygen, silicon and magnesium
...
Inner core, a solid sphere of nickel and iron
PDA [Preview]
Greyscale
English
Inner core, a solid sphere of nickel and iron

Figure 1: Three different versions of an adaptive diagram showing the structure of the earth.

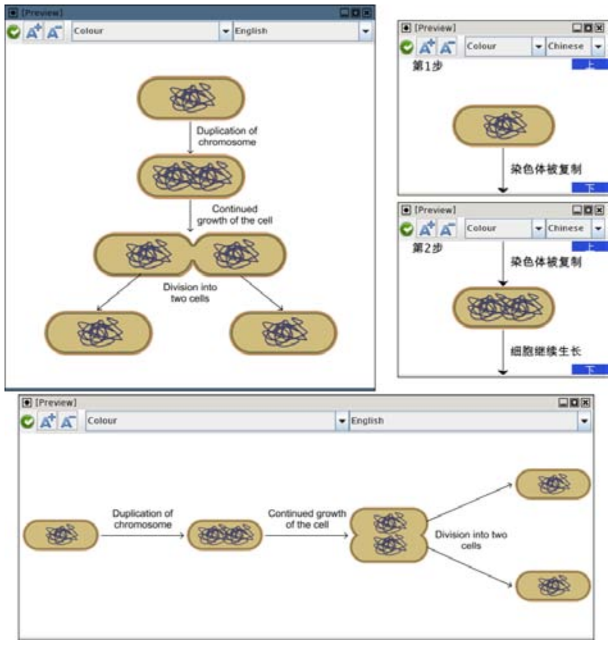

Figure 2: An adaptive diagram showing the steps involved in cell division.

tent in which a graphic element such as text can be determined at display-time. In terms of devices, we wanted to allow adaptation to at least a standard computer monitor, a mobile device with a relatively small screen, e-paper with a monochrome display (with very slow page refresh) and print. We wanted to cater for a variety of different languages and users with poor or no vision.

3.1 Constraint solving

Minor layout adjustment, in which the layout changes smoothly in response to changes in text or minor changes in the viewport dimensions, is handled in our tool by geometric constraint solving. Previously geometric constraint solving has been widely used to provide adaptive layout [1, 6, 8, 9, 17] and this is its primary role in our tool. The use of geometric constraint-solving is not unusual in diagram editors. However, previously this has been to facilitate subsequent editing rather than to specify how the layout should adapt. For example, the graphic editors Microsoft Visio[2] and ConceptDraw[3] provide persistent alignment and distribution placement tools. While important, this is a secondary role of constraint solving in our authoring tool.

However, there a variety of different constraint-solving technologies have been developed for graphical applications. We considered three approaches: one-way and linear arithmetic constraints which have been previously used to provide adaptive layout, and multi-way propagation-based constraints which have been used in graphical and GUI applications. We required a technique that was powerful enough to support the kinds of high-level constraints that naturally arise in diagrams, such as text boxes, alignment and distribution and we also wanted a technique that allowed the diagram's adaptive behaviour to be compiled into reasonably compact script.

One-way (or data-flow) constraints are the simplest, most widely used kind of constraint [19]. A one-way constraint is exactly like a formula in a spreadsheet cell. It has the form $x = f_x(y_1, ..., y_n)$ where the formula f_x details how to

[2]http://office.microsoft.com/visio
[3]http://www.conceptdraw.com/

compute the value of variable x from the variables $y_1, ..., y_n$. Whenever the value of any of the y_i variables changes, the value of x is recomputed, ensuring that the constraint remains satisfied. One-way constraints are versatile, simple to implement and can be solved extremely efficiently. They form the basis for constraint solving in Visio and ConceptDraw and for adaptive layout in many systems.

At first glance, one-way constraints seem ideal for our application since they support standard graphical constraints and they can be readily compiled into script. However, they have a significant limitation: constraint solving is directional and cyclic dependencies are not allowed, i.e., an attribute cannot be defined in terms of itself. For instance, consider the vertical alignment of three boxes A, B and C. This is naturally modelled by the one-way constraints

$$A.x \leftarrow L.x, \qquad B.x \leftarrow L.x, \qquad C.x \leftarrow L.x$$

where L is an "alignment guideline." When the author moves box B during editing the other boxes and alignment line will not follow it since the constraints only compute values for $A.x$, $B.x$ and $C.x$, and only as a result of changes to the value of $L.x$. A change to B effectively overwrites the formula that caused its position to depend on L.

If one-way constraints are used to specify how the layout should adapt, then the direction of constraint-solving is fixed by the application with viewport size and text box sizes driving the rest of the layout. However, this directionality may not be that expected by the author when editing the diagram. In addition user-studies have shown that users expect alignment and distribution constraints to behave symmetrically [20]. For instance, in the above example if box B is moved they will expect the other boxes and the alignment line to follow B.

We next investigated the use of linear-arithemetic constraint solving techniques [13]. These have also been used for adaptive layout. They are not directional and are, in essence, based on variable elimination techniques. Thus in the alignment example, changing the position of A, B, C or L will appropriately update the position of the other objects in the alignment. However, linear-arithmetic constraints are quite restricted and are not powerful enough to directly encode constraints like a text box should be large enough to contain its textual content. Furthermore, it is not that easy to compile linear arithmetic constraints into script if inequalities are allowed. We also experimented with a combination of one-way and linear-arithmetic constraints, but we found that certain combinations of layout and object size constraints that we wanted to support still resulted in cyclic dependencies.

Finally we considered the use of multi-way propagation based constraint solving methods [4, 18] in the authoring tool. Like linear-arithmetic constraints, multi-way constraints allow constraints to behave multi-directionally. All variables can potentially be output variables, so long as their value can be calculated from the values of the other variables. A multi-way constraint is specified by a set of one-way constraints which detail how the constraint can be solved for different choices of input variable. Thus in our example the alignment constraint would be specified by:

$$\begin{array}{llll} A.x \leftarrow L.x, & B.x \leftarrow L.x, & C.x \leftarrow L.x & \& \\ L.x \leftarrow A.x, & B.x \leftarrow A.x, & C.x \leftarrow A.x & \& \\ L.x \leftarrow B.x, & A.x \leftarrow B.x, & C.x \leftarrow B.x & \& \\ L.x \leftarrow C.x, & A.x \leftarrow C.x, & B.x \leftarrow C.x & \end{array}$$

Based on the values that are known the constraint solver efficiently constructs a "plan" to solve the other variables. The plan is simply a system of one-way constraints. Thus, changing the position of A, B, C or L will appropriately update the position of the other objects in the alignment.

We chose to use multi-way constraints because they support the standard graphical constraints and because, at least in theory, it is straightforward to compile them into script. The idea is that we generate a plan of how to adapt the layout given the viewport dimensions, text sizes and changes to interaction as input variables. Since this plan is a system of one-way constraints it is readily compiled into script.

3.2 The Authoring Tool

In order to make the tool easy to learn to use, we decided that it should, as far as possible, extend the construction model underlying commonly used diagramming tools such as Microsoft Visio or OmniGraffle[4]. The obvious minimal extension to this construction model is for the author to explicitly construct a number of alternate versions of a diagram, with a separate version for each possible adaptive layout. The appropriate version would then be chosen based on the viewing environment and previous user interaction. This, however, is not very practical since there may be an extremely large number of these alternate versions.[5]

Thus, we felt that the tool's design must not require the author to explicitly construct all alternate versions but rather allow them to be constructed automatically whenever possible and only when needed. A key observation underlying this automatic construction is that style, text, object representation and overall layout are largely orthogonal. For instance, essentially the same changes are required to modify a narrow and wide version of a diagram from English text to French text or from colour to monochrome. Thus, in our tool the author can specify alternate layouts, alternate representations for diagram components, alternate styles and alternate text. The resulting space of different versions is defined implicitly to be the cross product of these different alternatives.

Layout configurations: To specify structurally different layouts the author must explicitly create a separate alternate layout, called a *configuration*, for each structurally different layout. Different configurations cater for major changes in the layout such as its orientation, the form of the diagram, the use of indirection in the diagram and how animation is represented. Configurations are the core of the authoring tool and behave similarly to the drawing canvas in a standard graphic editor. However, unlike in a traditional editor where multiple canvases represent different diagrams, multiple configurations are alternate layouts for the same logical diagram.

Another important difference between a configuration in our tool and a drawing canvas in a traditional diagram editor is that a configuration will automatically adjust the layout to cater for different text, style, choice of diagram component representation or viewport dimensions. The author

[4]http://www.omnigroup.com/applications/OmniGraffle/

[5]For example, if we have four different devices, two target languages and consider colour-blind users then we have sixteen different versions. If we also consider user-controlled zooming and creating a different version for each possible combination of expanded/collapsed sub-diagrams, then the number of versions grows exponentially.

specifies how this layout adjustment should occur by using constraint-based placement tools which place persistent layout relationships between the configuration elements. These relationships will be maintained during adaptation and also during subsequent editing until the author removes them.

Since the authoring tool is designed for diagrams utilising grid-based layout, it provides horizontal and vertical alignment and distribution tools. Text boxes that compute either the smallest height required to fit the text given a fixed width, or the minimum width for a fixed height, are also provided. In addition, raster images included in a configuration will maintain their intrinsic aspect ratio if resized in response to layout adaptation.

Text master: As the choice of textual content is usually orthogonal to the layout of a diagram, text is specified separately from the configurations in a text master. For each piece of text in a configuration the author can supply different versions, one for each possible display language. Furthermore, the author can optionally supply for a particular display language alternate versions of the text, typically the full text and an abbreviated version. The system will choose, based on the available space for layout, the best alternate version of text to use for the current display language. This choice is global rather than made individually for each piece of text. Thus, if the abbreviated version is chosen, all text will be abbreviated.

Style master: Since the choice of styling is orthogonal to the layout of a diagram, it is specified separately from the configurations in a style master. The styling system consists of two parts: a set of style classes, each of which is a set of property–value pairs (as in CSS), and a *style palette*, which is a repository of style values that can be referenced by the style class rules. The entries in the palette can be colours, patterns or even numerical values.

At presentation time, a particular style mode is applicable. This style mode is chosen by the user agent based on the capabilities of the medium and the viewer, and is currently limited to *colour*, *greyscale* and *monochrome*, although this could readily be extended to handle cases such as different kinds of colour blindness or to target a tactile diagram printer. The style palette can be used to specialise styling for particular modes. For example, an entry in the style palette may be set to the colour red for the colour mode, while for greyscale and monochrome the entry can be set to a shade of grey and black, respectively. The authoring tool provides reasonable defaults to map colour values to greyscale and monochrome, which can be overridden by the author if desired.

Objects: Objects, that is graphic elements and associated placement constraints that form a logical diagram component, are often re-used in different configurations. To facilitate re-use, the author can place an object in a *object master* and use instances of it in different configurations. Changes made to the master object, such as changing a node from being text in a circle to text in a rectangle are automatically propagated to all instances.

Objects can be parametric in the choice of text, facilitating construction of reusable objects that differ only in their text content. The guidelines representing alignment constraints in an object can be attached to other guidelines and objects when an object instance is placed in a configuration. This allows the layout of the object to adjust to its context in quite flexible ways, such as having multiple objects be sized to the same width by being attached to a distribution, or by exporting attachment points which arrows in a configuration can have their endpoints attached to.

Objects are allowed to have alternate representations. A typical use is to provide a compact and expanded representation for the object. Currently, the choice of which representation to use is controlled by explicit user interaction. Also, the tool does not allow nesting of object instances.

Interaction: Viewer-interaction can be used to control diagram adaptation. The author specifies this by using global, numerical state variables whose values are changed by user interaction and whose values affect the choice of configuration, style class, and object representation. Any object in the diagram can be annotated with a list of actions to perform when an event occurs on that object (such as moving the mouse cursor over it, or clicking it). These actions modify the state variables (by setting them to particular values, or the result of evaluating an expression) and in so doing, cause the diagram to adapt.

Each instance of an object master with alternate representations has an *alternate selection policy*, which is simply an expression over the state variables that, when evaluated, gives the index number of the alternate representation to use. Style class definitions can also be conditional based on state variables, thus allowing diagram styling to react to user interaction. This can be used to, for example, highlight parts of a diagram when clicked or hovered over by the mouse pointer.

Previewing: The authoring tool allows the author to specify multiple preview windows, which show how the diagram will appear in different viewing environments. As the diagram is edited, the preview windows automatically update their view of the diagram. Each preview window has its own viewing environment explicitly set by the author and has its own history of user-interaction. These viewing environments can be named, saved and shared between different authoring sessions. The tool provides a number of default preview windows corresponding to standard viewing environments: standard computer monitor, mobile device, e-paper and a monochrome printing device.

An example editing session is shown in Figure 3. Here, three of the master objects and the three configurations that make up the example from Figure 1 are shown.

3.3 Choosing the version

When the diagram is to be displayed the system must choose which combination of configuration, object alternates, text and style is used to create the version. We distinguish between *environment driven* choices for which the viewing environment specifies a unique choice, such as the display language; *interaction driven* choices for which the choice is also unique but which is the result of user interaction, such as whether an object is expanded or collapsed; and *search driven* choices, such as which configuration to use and which level of text abbreviation, for which the system must try different possibilities in order to determine the best choice.

One of the most difficult decisions in the authoring tool design was determining how to allow the author to specify the preference order on versions. One approach would have been to require the author to give an explicit objective function measuring "goodness" and for the system to choose the layout maximizing this function. Although very general, we felt this would be difficult for the author. The

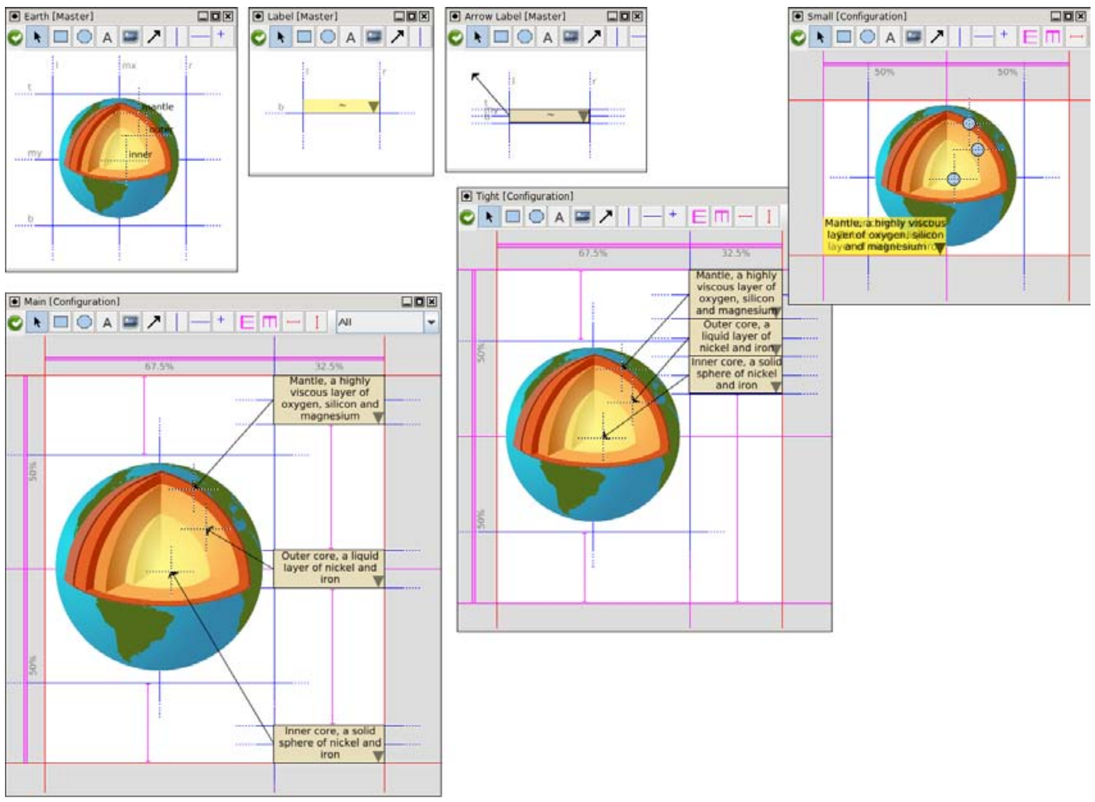
Earth [Master]
Label [Master]
Arrow Label [Master]
Small [Configuration]
Tight [Configuration]
Main [Configuration]
mantle
outer
inner
All
67.5%
32.5%
50%
Mantle, a highly viscous layer of oxygen, silicon and magnesium
Outer core, a liquid layer of nickel and iron
Inner core, a solid sphere of nickel and iron

Figure 3: An editing session for the example in Figure 1, showing three master objects (*Earth*, *Label* and *Arrow Label*) and three configurations (*Main*, *Tight* and *Small*).

other approach is for the author to provide a preference ordering on the different possible versions. We chose to take this approach.

The configurations have an explicit ordering from most preferred to least preferred. The author controls this by rearranging the configurations in an ordered list. Each configuration has a set of text content and styles for which it is compatible. By default a configuration is compatible with all text content and styles but the author can restrict the choice. This means that the author can create configurations that are language- or style mode-specific.

The placement tools implicitly add compatibility restrictions on the configuration. In distribution constraints, the alignment lines in the distribution cannot change their order, and graphical objects attached to a guideline must all be positioned accordingly. The author can add additional *tests* to the configuration to check that the layout is reasonable. There is a test for a minimum horizontal or vertical separation which allows the author to ensure that two objects do not overlap. Tests are passive: the layout is computed and then the test is evaluated.

The diagram version is chosen dynamically. The system examines those versions that are compatible with the environment and interaction driven choices. There is a total ordering on these versions from most preferred to least preferred. The primary ordering is given by the preference ordering on the configurations, with a secondary ordering given by the ordering on the alternate choices of text for the display language. The versions are created and examined in descending order of preference. The first version created that is *valid*, i.e. for which the layout constraints and layout tests are satisfied, is chosen.

One issue is what to do in the case that user interaction changes the object representation. This may mean that a more preferred version that was previously invalid becomes valid because of this new choice of representation. One approach would be to change versions. We felt that this might be disturbing to the user, and also has a high overhead since it means that the choice of diagram version has to be reconsidered after virtually all user interaction. Instead we have chosen to keep the current version until it becomes invalid. However, changes to the viewport dimensions or text size will lead to a total re-computation of the preferred version.

3.4 Compiling to SVG

One of the goals of our authoring tool was that diagrams constructed with it should be able to be compiled into a widely-used interactive graphics format. This naturally led us to target scripted SVG. In order to demonstrate the feasibility of compiling an adaptive diagram to SVG, we took the example from Figure 3 and compiled it by hand, making sure that the structure of the resulting code mirrored our adaptive diagram model.[6] The compilation was purely mechanical; no special optimisations were used in the hand-compiled version.

As discussed previously, a fundamental observation is that while multi-way constraints are used during editing, at pre-

[6]The complete compiled example can be found at http://mcc.id.au/2008/05/earth.svg.

```
function resize() {
    // The canvas has been resized, so perform a full layout on
    // each configuration to see which is satisfiable.
    for (var i = 1; i <= 3; i++) {
        if (layoutConfig(i)) {
            switchConfig(i);
            break;
        }
    }
}

function switchInstance(instance, master) {
    if (instance == Config2Label1) {
        // (...get image position...)
        satisfiable = layoutConfig2Label1Onwards
          (imageWidth, imageTop, imageHeight);
    } else if (instance == Config2Label2) {
        // (...get image and label 1 position...)
        satisfiable = layoutConfig2Label2Onwards
          (imageWidth, imageTop, imageHeight, label1Bottom);
    } else if ...
    // (...if not satisfied, check other configurations...)
}

function layoutConfig2() {
    // (...position the <image> element...)
    if (imageTop < 0 || imageHeight > canvasHeight) {
        return false;
    }
    return layoutConfig2Label1Onwards
              (imageWidth, imageTop, imageHeight);
}

function layoutConfig2Label1Onwards(...) {
    // (...position the top-most text label instance
    // and determine its satisfiability...)
    return label1Satisfiable &&
        layoutConfig2Label2Onwards
          (imageWidth, imageTop, imageHeight, label1Bottom);
}
```

Figure 4: Layout functions from the compiled version of the earth structure adaptive diagram.

sentation time the number of variables whose values can change due to user interaction is limited. Consequently, script to implement relayout can be split into a number of functions, where each function corresponds to executing the constraint plan for a given set of input variables. For example, consider a guideline with three rectangles attached to it. Three constraints exist, equating each rectangle's $left$ variable with the guideline variable G, as illustrated by Figure 5(a), and there are four possible plans to solve the system of constraints. Once one variable is chosen to be an input variable, however, there exists only a single plan. Figure 5(b) shows the constraint graph after $R2.left$ is chosen to be the input variable, for example because its position is determined by the canvas width when the canvas is resized.

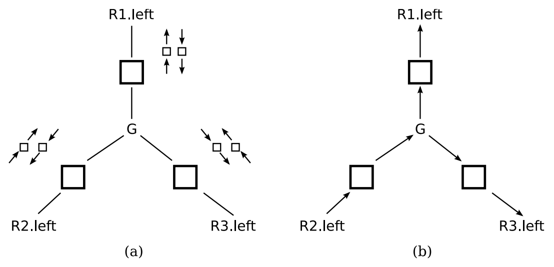

Figure 5: Choosing a constraint plan when compiling: (a) multi-way constraints at design time, and (b) a plan to solve the constraint system when $R2.left$ is the input variable.

```
function set(varname, val) {
    stateVariables[varname] = val;
    if (varname == 'showMantle' && currentConfig == 3)
        switchInstance
            (Config3Label1, val ? Label : EmptyLabel);
    else ...
}

<g id='Configuration3'>
    <ellipse class='hotspot' rx='8' ry='8'
        onmouseover='set("showMantle", 1)'
        onmouseout='set("showMantle", 0)'/>
    ...
</g>
```

Figure 6: Interaction function from the earth structure diagram, and part of the graphical content comprising the third configuration.

Figure 4 is a heavily condensed version of some layout functions from the compiled example. In the second configuration of the diagram, there are four things that can change, and must then cause relayout (and a possible switch of configuration): canvas size, and the selected alternative for each of the three text label instances. Since the position of the third text label instance depends on the position of the second text label instance (and the second depends on the first), functions exist to lay out the configuration starting with one of the text labels. For example, *layoutConfig2Label1Onwards* positions the first text label and then goes on to call *layoutConfig2Label2Onwards* to handle the remaining text labels. Each of these functions returns a boolean to indicate whether the layout was satisfiable, which is ultimately checked in the top level layout functions (*resize* and *switchInstance*) to see if the next configuration should be tried.

Since SVG lacks a suitable templating mechanism, instances of master objects are implemented by having the graphical content of each master object stored in the definitions section (the <`defs`> element), and using script to clone these sub-trees when instantiating them. SVG also does not have native support for text boxes whose width is determined by their height, so code to perform this text wrapping must also be included in the compiled diagram.

Other aspects of the diagram adaptation map more naturally to elements of the SVG document: the style class system (including the palette and style modes) correspond to a block of CSS that utilises Media Queries[7] [11]; interaction maps to standard SVG event handlers that manipulate script variables and then switch instance alternatives as appropriate, as shown in Figure 6; and the strings in the text master correspond to <`text`> elements in the document which are re-used when a given text string is referenced in the diagram.

4. DISCUSSION

We have completed a prototype implementation of the authoring tool. It is written in a combination of Java and ECMAScript, and comprises approximately 17,000 lines of code. We have used it to construct a number of adaptive diagrams, including the examples shown in this paper. In this section we present an analysis of its design with respect to our five original design requirements and suggest various possible extensions and improvements.

[7]Note however that media queries would not be able to handle non-colour style modes, such as "tactile."

Learnability: Our design decision to base authoring model on the diagram construction model provided in existing diagramming tools has worked well. Configurations, placement constraints, object masters, and the styling system extend familiar concepts. Alternate objects, text master and the specification of interaction are more unfamiliar, but we believe reasonably straightforward to understand.

Predictability: The second requirement was that the adaptive behaviour of a diagram should be understandable and predictable. Requiring the author to explicitly construct a new configuration if the layout or form of the diagram differs substantially from the other configurations aids this. Layout adjustment within a single configuration is specified with standard placement tools whose behaviour is quite predictable. Allowing multiple preview windows works well, allowing the author to immediately see the impact of their changes in different viewing environments.

Low level support: Another requirement was that the resulting adaptive diagrams can be compiled into a reasonably compact representation using SVG with scripting. Currently, we have done this by hand for a simple example to verify that it is possible and are now building a compiler based on this translation.

Usability and generality: The final two requirements were that the tool allow the user to readily author a wide variety of adaptive behaviours for the target class of grid-like diagrams. The simplest kind of adaptive behaviour such as adjusting the layout to a different display languages, shrinking whitespace, or adapting to a monochrome viewing device happen essentially automatically so long as the author uses placement tools to place the objects in a configuration. This is a considerable improvement over current diagram authoring tools. Other kinds of adaptive behaviour, such as interactive collapsing and expanding of objects require more effort from the author. However, other diagramming tools do not support this, and at best allow the author add script to the exported SVG *post facto*.

Limitations: Our experience suggests that the tool design is more than adequate for many examples. However, it does have limitations. One of the main issues is restrictions on what can be placed in the object master which limit re-use of diagram components. Currently objects in the object master cannot contain instances of another object in the object master making it impossible for the author to reuse sub-components of master objects as well as the objects themselves. Furthermore, alternate representations are not allowed in the object master, instead these must be specified again in each configuration. For example, when we attempted to construct an organization chart which allowed the user to collapse and expand sub-trees in the chart, we could only reuse the individual nodes in the chart but not the sub-trees. We plan to remove these restrictions on what can be placed in the object master in the next version of the tool. A related limitation is that currently alternative representations for objects cannot be nested. This also restricts the amount of reuse. However, we believe that such nested alternatives might be difficult for users to understand and construct. Furthermore, they complicate specification of user interaction and choice of versions.

There are a number of other limitations which reflect a lack of implementation rather than being consequences of poor design choices. First, while the tool supports discrete animation (in the form of interaction-controlled alternate selection) it does not support continuous animation. Also, the tool does not support automatic generation of repeated elements–the author must explicitly construct a new configuration for each number of repeated elements. Third, the authoring tool does not support adaptation to dynamic content. Adding support for dynamic textual content or choice of image would be relatively simple. However, it would be a major extension to add support for dynamic structural layout, as required for instance to display a dynamically generated organization chart or bar chart. And finally the tool does not provide automatic support, such as layout wizards, for constructing standard diagram types such as maps, bar charts or organization charts from external data. At present the construction of a bar chart is very tedious, requiring the user to explicitly construct the axis and bars themselves. We plan to add automatic support for construction of bar charts and organization charts to our tool. This will also generate reasonable default adaptive behaviours.

5. CONCLUSION

As far as we are aware, this is the first research into how to design a diagramming authoring tool for authoring diagrams that can adapt to their viewing environment including user requirements. We believe our prototype tool demonstrates that our adaptive diagram model allows the construction of adaptive diagrams whose layout is grid-like, and that the orthogonality of the adaptive behaviour specification (i.e., specifying layout, style, text and interaction independently) reduces the effort required to author a given diagram. We have also demonstrated how an adaptive diagram authored with the tool may be compiled into scripted SVG, so that it can be used in today's web browsers, and we believe that this is also true of all adaptive diagrams expressible with our model.

The tool was designed to cater for diagrams whose layout is grid-like. Supporting diagrams whose layout is not grid-like such as force directed graph layout would require more powerful constraint solving capabilities. This would then make compilation into SVG with script more difficult, unless the script had access to powerful layout engines.

In the future, we plan to extend the authoring tool to provide more flexible layout placement tools (such as distributions that distribute space rather than positions), to allow instantiation of master objects within master objects themselves so that hierarchies of re-use are possible (and to investigate the difficulties with the specification of alternate object representations that results from this), to add a compilation-to-SVG feature to the tool, and to look at providing high-level wizards or editors for specific diagram types such as charts, which would otherwise be tedious to construct with the tool. In addition, we intend to perform user testing on the authoring tool to validate our design decisions and to improve in the interface.

6. REFERENCES

[1] A. Borning, R. K.-H. Lin, and K. Marriott. Constraint-based document layout for the web. *Multimedia Syst.*, 8(3):177–189, 2000.

[2] B. Bos, T. Çelik, I. Hickson, and H. W. Lie. Cascading Style Sheets level 2 revision 1 (CSS) 2.1 specification. http://www.w3.org/TR/CSS21/, July 2007.

[3] J. Ferraiolo, J. Fujisawa, and D. Jackson. Scalable Vector Graphics (SVG) 1.1 specification. http://www.w3.org/TR/SVG11/, January 2003.

[4] B. N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Commun. ACM*, 33(1):54–63, 1990.

[5] N. Hurst, K. Marriott, and P. Moulder. Cobweb: a constraint-based web browser. In *ACSC '03: Proceedings of the 26th Australasian computer science conference*, pages 247–254, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[6] C. Jacobs, W. Li, E. Schrier, D. Bargeron, and D. Salesin. Adaptive grid-based document layout. *ACM Trans. Graph.*, 22(3):838–847, 2003.

[7] C. Jacobs, W. Li, E. Schrier, D. Bargeron, and D. Salesin. Adaptive document layout. *Commun. ACM*, 47(8):60–66, 2004.

[8] M. Jourdan, N. Layaïda, C. Roisin, L. Sabry-Ismaïl, and L. Tardif. Madeus, an authoring environment for interactive multimedia documents. In *MULTIMEDIA '98: Proceedings of the sixth ACM international conference on Multimedia*, pages 267–272, New York, NY, USA, 1998. ACM.

[9] M. Jourdan, C. Roisin, and L. Tardif. Constraint techniques for authoring multimedia documents. *Constraints*, 6(1):115–132, 2001.

[10] R. Lewis. Authoring challenges for device independence. http://www.w3.org/TR/acdi/, September 2003.

[11] H. W. Lie, T. Çelik, and D. Glazman. Media queries. http://www.w3.org/TR/css3-mediaqueries/, June 2007.

[12] J. Lumley, R. Gimson, and O. Rees. A framework for structure, layout & function in documents. In *DocEng '05: Proceedings of the 2005 ACM symposium on Document engineering*, pages 32–41, New York, NY, USA, 2005. ACM.

[13] K. Marriott and S. S. Chok. QOCA: A constraint solving toolkit for interactive graphical applications. *Constraints*, 7(3-4):229–254, 2002.

[14] K. Marriott, B. Meyer, and L. Tardif. Fast and efficient client-side adaptivity for SVG. In *WWW*, pages 496–507, 2002.

[15] C. McCormack, K. Marriott, and B. Meyer. Adaptive layout using one-way constraints in SVG. In *SVG Open 2004: Proceedings of the third annual conference on Scalable Vector graphics*, September 2004.

[16] K. Moore. Every page is different: a new document type for commercial printing. In D. C. A. Bulterman and D. F. Brailsford, editors, *ACM Symposium on Document Engineering*, page 2. ACM, 2006.

[17] J. van Ossenbruggen, J. Geurts, F. Cornelissen, L. Hardman, and L. Rutledge. Towards second and third generation web-based multimedia. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 479–488, New York, NY, USA, 2001. ACM.

[18] B. Vander Zanden. An incremental algorithm for satisfying hierarchies of multiway dataflow constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(1):30–72, 1996.

[19] B. Vander Zanden, R. Halterman, B. A. Myers, R. McDaniel, R. Miller, P. Szekely, D. A. Giuse, and D. Kosbie. Lessons learned about one-way, dataflow constraints in the garnet and amulet graphical toolkits. *ACM Trans. Program. Lang. Syst.*, 23(6):776–796, 2001.

[20] M. Wybrow, K. Marriott, L. McIver, and P. J. Stuckey. Comparing usability of one-way and multi-way constraints for diagram editing. *ACM Trans. Comput.-Hum. Interact.*, 14(4):1–38, 2008.