Flexible Graph Layout for the Web

Trevor Hansen, Kim Marriott & Bernd Meyer
School of Computer Science and Software Engineering
Monash University
Clayton, Victoria 3168, Australia
{marriott,berndm}@csse.monash.edu.au

Peter J. Stuckey
Dept. of Computer Science and Software Engineering
University of Melbourne
Parkville, Victoria 3052, Australia
pjs@cs.mu.oz.au

September 14, 2001

Corresponding Author Information	
	Peter J. Stuckey
Address	Department of Computer Science and Software Engineering
	University of Melbourne 3010, Australia.
Email	pjs@cs.mu.oz.au
Phone	+613-8344-9155
Fax	+613-9348-1184

Abstract

More powerful personal computers and higher network bandwidth has meant that graphics has become increasingly important on the web. Graph-based diagrams are one of the most important types of structured graphical information. Here we demonstrate how XML can be used as basis for contents-based delivery of graph-based diagrams. The main distinguishing feature of our approach is that it separates style and content of diagrams in the same way as (XML-based) markup languages for textual information do: The diagram itself is marked-up according to its logical structure and its visual appearance is defined via attached style-sheets. Such an approach poses interesting challenges for the browser component, because it requires automatic layout of complex diagrammatic information that takes stylistic constraints into account. We present a prototype system for our approach comprised of three main components: A contents-based markup language, GXML, for specifying graph-based diagrams, a style sheet language, GXSL, for such diagrams and a browser that can display styled graphs from this information.

Abstract

More powerful personal computers and higher network bandwidth has meant that graphics has become increasingly important on the web. Graph-based diagrams are one of the most important types of structured graphical information. Here we demonstrate how XML can be used as basis for contents-based delivery of graph-based diagrams. The main distinguishing feature of our approach is that it separates style and content of diagrams in the same way as (XML-based) markup languages for textual information do: The diagram itself is marked-up according to its logical structure and its visual appearance is defined via attached style-sheets. Such an approach poses interesting challenges for the browser component, because it requires automatic layout of complex diagrammatic information that takes stylistic constraints into account. We present a prototype system for our approach comprised of three main components: A contents-based markup language, GXML, for specifying graph-based diagrams, a style sheet language, GXSL, for such diagrams and a browser that can display styled graphs from this information.

keywords: graph layout, constraints, style sheets, World Wide Web.

1 INTRODUCTION

With the advent of increasingly more powerful personal computers and higher bandwidth connections the web has become a medium that relies more and more on graphical communication. Traditionally, most of the graphical information exchanged on the web is in the form of low-level pixel-based encodings such as GIF or JPEG formats, but it is clear that for many applications this is inadequate. Not only does such an encoding waste bandwidth, it does not support searching and does not allow manipulation of the graphical contents on the client side. Worse, it makes it virtually impossible for a client to adapt graphics to different viewing conditions, such as very small displays on PDAs or mobile phones.

The arguments for a separation of structure from style, such as adaptability to viewing conditions, searchability, ability to easily generate output from databases etc., are well known. In fact, this separation of concern is one of the fundamental issues in the design of languages for web documents and is manifest in almost all web standards from the HTML/CSS [1] combination to XML/XSL [6, 23].

Recent graphics standards, such as SVG [12] and VRML [20] have addressed this issue and have introduced structured high-level representation of graphics. However, SVG and VRML still do not use a contents-based representation, but simply a vector-based or object-oriented graphics representation consisting of high-level visual entities, such as filled polygons, instead of pixel-based representations.

In this paper we show how XML can be used as basis for delivery of high-level graphics information that is truly contents-based. The basic idea is that, analogously to the use of XML for textual information, we can use XML to define a high-level graphics markup language for each class of diagrammatic languages and use either dedicated browsers to display concrete visualizations of this data or use a dedicated (possibly XSL-based) preprocessor to convert the XML-based description into concrete graphics information in an appropriate vector-based format, such as SVG, which can be displayed by standard browsers.

In this paper we focus on graph-based diagrams, such as class hierarchies or state-transition diagrams, which are one of the most important types of structured graphical information. They occur in almost every technical discipline as well as in many non-technical contexts. On the web a particularly important application is the visualization of web-site structures (site maps). We demonstrate how XML can be used as basis for contents-based delivery of such graph-based diagrams.

There have been previous proposals for graph modelling languages, using both XML [18] and non-XML languages [19]. The focus there has been on providing an interchange format for graphs. None of them provide a language for describing graph styles or the ability to separate content information from layout information. Nor do they address the problem of how to display and browse the graphs. Significant other differences arise in the treatment of hierarchical graphs, which have to be decomposed into flat graphs in the above treatments, but are an integral part of the "document" structure in our markup language.

We have developed a prototype system whose architecture consists of three main components: A contents-based markup language, GXML, for specifying graph-based diagrams, a style sheet language, GXSL, for such diagrams and a post-processor or browser that can display styled graphs from this information. One interesting feature of the system are the powerful sub-graph matching constructs provided in GXSL. These are required because unlike textual documents which are naturally tree-structured, diagrams, and in particular graphs, are not.

Clearly, any system based on such an architecture crucially depends on a component which computes a concrete layout for the graph from the structure and style information. To illustrate the importance of layout, consider Figure 1 and Figure 2.[1] Both visualize the same web site: information on a computer science subject numbered 141 which has various pages relating to organization, news and projects. While Figure 2 fairly clearly exhibits its structure, Figure 1 only does so inadequately.

It is desirable to support at least some elementary interaction in a graph browser. At the very least the browser should allow the viewer to move nodes and collapse/expand subgraphs in order to support hierarchical exploration of the graph. This means that the browser must also support re-layout of the graph during interaction.

The automated drawing of graphs is a conceptually difficult and complex (computationally hard) task, and a whole sub-field of data visualization is devoted to the development of graph drawing algorithms [14]. An excellent overview of the current state of the art can be found in [10]. Specialized algorithms have been developed for many different specific types and styles of graphs, in particular trees, directed acyclic graphs, orthogonal graphs and several domain-dependent forms. Unfortunately, the style of the final layout in existing graph layout algorithms is almost always "hard-wired", e.g. a layout algorithm for orthogonal graphs is not useful when a tree has to be displayed.

However, in the suggested architecture, the separation of graph structure from layout style is a core concept and the layout mechanism must, for example, be able to satisfy additional specific layout requirements that arise from a graph style sheet, such as some nodes must be at the center of the graph, particular types of nodes must be

[1] All figures in this paper were rendered by our GXML previewer prototype from GXML files.

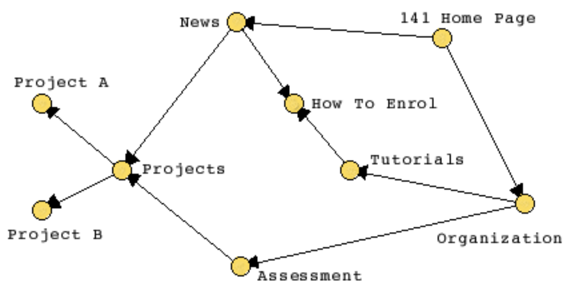

Figure 1: A web site diagram with labelled circular nodes.

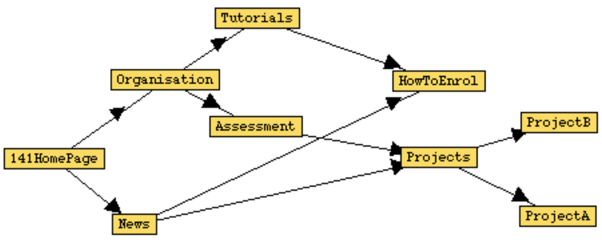

Figure 2: A web site diagram with labelled rectangular nodes.

aligned horizontally, etc. Thus, most existing graph layout algorithms do not support our architecture.

We will therefore describe a new kind of layout method based on simulated annealing which can be used to support our architecture since it allows flexible layout in the presence of user-defined style constraints.

The remainder of the paper is structured as follows: Section 2 presents the graph markup language (GXML) and Section 3 discusses the graph style sheet language (GXSL) and its relation to GXML. Following this, Section 4 presents the structure of our prototype system to process these languages and discusses possible alternative architectures. Finally, we sketch a flexible layout method that can accommodate the user-defined style constraints in Section 5. Section 6 concludes.

2 GRAPH MARKUP LANGUAGE (GXML)

This section describes the structure-based Graph Markup Language (GXML), which is specified using XML. The core of GXML is straight-forward: XML-defined tags describe the logical components of the graph, i.e. its nodes, edges, and node and edge types (classes). A graph can be hierarchically structured by using subgraphs as logical nodes in another graph.

Not only can GXML be used for a purely structural description of a graph, but it is also possible to specify concrete layout information by defining coordinates, sizes, stroke weights, bitmaps for nodes and edges, etc. However, it is important that these elements can be left undefined or partially defined, since it is the task of the browser/preprocessor to compute a concrete visualization from the GXML code and the style sheet. We allow the concrete layout information to be part of GXML so that we can use it uniformly as the representation of graphs before and after layouting.

Figure 3 gives the GXML code for the example graph in Figure 2. We will briefly discuss the logical aspects and the layout aspects of GXML in turn.

Each GXML file contains one or more `<GRAPH>` tags each of which is a list of `<NODE>` tags and `<EDGE>` tags with the obvious meanings. `ID` attributes of nodes are used as references in edge definitions, edges carry an attribute that defines whether they are `DIRECTED` and all objects can use a `CLASS` attribute that defines a conceptual kind of node, edge or graph. This attribute can be queried in style sheets to allow different layout conditions for different kinds of nodes, edges or graphs. In addition the standard `ALT` and `IMG` attributes can be used with graphs for browsers that do not support GXML. Nodes, edges and graphs can all contain a `<LABEL>` element which defines a textual label to be displayed with the respective element.

A slightly more involved aspect of the logical GXML structure is the definition of hierarchical graphs. This is modeled in the following way: A `<GRAPH>` tag can recursively contain other `<GRAPH>` tags. A subgraph must define a `<PORTLIST>` element which associates an abstract unique port identifier with each node of the subgraph that is incident to a node outside of this subgraph. The idea is that the thus defined ports are used as the attachment point for edges outside this subgraph. It is an error for an arc to directly connect a node from outside the subgraph with a node within the subgraph. In a fully expanded view such edges can then directly be routed to the associated node, while in a collapsed view the non-expanded subgraph is represented by some primitive shape with only these ports on its outside and the edges are routed to the ports. Figure 4 illustrates this with a collapsed version of our example graph. Accordingly, each subgraph has a `VISIBLE` attribute that can take the values `EXPANDED` and `COLLAPSED` with the obvious meanings, `HIDDEN` which prevents the subgraph and its associated edges from being displayed and `OUTLINED` which is identical to `EXPANDED` but requests an additional borderline drawn around the subgraph in order to make the hierarchy apparent. The ports are invisible except in `OUTLINED` mode where the external edge is routed to the port and an additional edge from the port to the internal reference node is added.

We also allow arbitrary *non-hierarchical* grouping of nodes in a graph into possibly overlapping sub-graphs using the `<SUBGRAPHS>` element which in turn consists of an arbitrary number of `<SUBGRAPH>` elements. Each `<SUBGRAPH>` element has a `<NODEREF>` element listing the IDs of the nodes in the subgraph. Unlike hierarchical subgraphs they cannot be expanded or collapsed independently of the graph they occur within. Consequently there is no need for connection points. There are two reasons why subgraphs may have to be used: To display outlines around non-hierarchical groups of nodes and to provide additional structural information as input to the style sheet processor so that, for example, elements in the same subgraph can be displayed tightly clustered.

In the following we describe the elements that are used to specify concrete layout

```
<!DOCTYPE GML SYSTEM "gxml.dtd">
<GRAPH>
  <NODE ID ="root"> <LABEL> 141 Home Page </LABEL> </NODE>
  <EDGE FROM="root" TO="ap1" DIRECTED="TRUE"></EDGE>
  <EDGE FROM="root" TO="up1" DIRECTED="TRUE"></EDGE>
  <NODE ID="update"> <LABEL> News </LABEL> </NODE>
  <EDGE FROM="update" TO="pp1" DIRECTED="TRUE"></EDGE>
  <EDGE FROM="update" TO="ep2" DIRECTED="TRUE"></EDGE>
  <EDGE FROM="sp1" TO="pp1" DIRECTED="TRUE"></EDGE>
  <GRAPH>
    <LABEL> Organization </LABEL>
    <PORTLIST> <PORT ID="ap1" ASSOC="about"/>
               <PORT ID="sp1" ASSOC="assess"/>
               <PORT ID="ep2" ASSOC="ep1"/>
    </PORTLIST>
    <NODE ID="about"> <LABEL> Organization </LABEL> </NODE>
    <EDGE FROM="about" TO="assess" DIRECTED="TRUE"></EDGE>
    <EDGE FROM="about" TO="tp1" DIRECTED="TRUE"></EDGE>
    <NODE ID="assess"> <LABEL> Assessment </LABEL> </NODE>
    <GRAPH>
      <LABEL> Tutorials </LABEL>
      <PORTLIST> <PORT ID="tp1" ASSOC="tutes"/>
                 <PORT ID="ep1" ASSOC="enrol"/>
      </PORTLIST>
      <NODE ID="tutes"> <LABEL> Tutorials </LABEL> </NODE>
      <EDGE FROM="tutes" TO="enrol" DIRECTED="TRUE"></EDGE>
      <NODE ID="enrol"> <LABEL> How to Enrol </LABEL> </NODE>
    </GRAPH>
  </GRAPH>
  <GRAPH>
    <LABEL> Projects </LABEL>
    <PORTLIST> <PORT ID="pp1" ASSOC="ps"/> </PORTLIST>
    <NODE ID="ps"> <LABEL> Projects </LABEL> </NODE>
    <EDGE FROM="ps" TO="pA" DIRECTED="TRUE"></EDGE>
    <EDGE FROM="ps" TO="pB" DIRECTED="TRUE"></EDGE>
    <NODE ID="pA"> <LABEL> Project A </LABEL> </NODE>
    <NODE ID="pB"> <LABEL> Project B </LABEL> </NODE>
  </GRAPH>
</GRAPH>
```

Figure 3: GML description of the hierarchical graph shown in Figures 1, 2 and 4.

information for a graph: Virtually all non-edge elements have optional `X` and `Y` attributes, which can be used to suggest the x- and y-position of the object. For labels we can define `FONT` attributes such as `COLOR`, `SIZE`, and `FACE`.

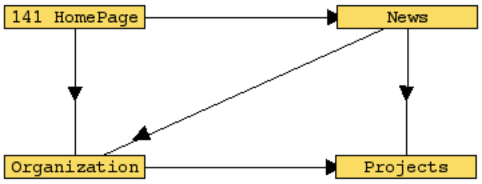

Figure 4: A reduced web site diagram to accommodate large fonts.

All nodes and subgraphs can have an `<OBJSHAPE>` attached to them which defines how this node is displayed. It is either a predefined primitive shape, such as `<CIRCLE-NODE>`, `<RECTANGLE-NODE>` or `<ROUNDTANGLE-NODE>`, or an arbitrary graphics shape defined by SVG [12] code inside an `<SVG>` element. In analogy, an edge can have a `<CONNECTOR>` and an `<ARROWHEAD>` subtag to define the visual appearance of the edge itself and its direction indicator (if applicable). Again, both of these tags can either contain basic predefined elements, such as `<STRAIGHTLINE>`, `<CURVEDLINE>` or `<SIMPLEARROW>` or an arbitrary `<SVG>` element.[2]

Each element that carries a label can make use of a `LABELPOS` attribute that suggests a preferred placement of the label relative to the object being labeled, i.e. north, north-east etc. Edges can additionally use a `PATHFRACTION` attribute indicating how far along the edge to place an edge label.

GXML also supports layout description on a more abstract and versatile level by means of the `<CONSTRAINT>` tag. As we have seen in the motivating example, the designer will often wish to provide layout information in terms of desired geometric relationships between elements of the graph, for example alignment or relative distances. This can be used to convey information about aesthetically desirable layout and also to capture additional layout requirements arising when the diagram has a richer semantic structure.

Our approach is to use constraints for specifying layout. A constraint is simply a statement of a relation (in the mathematical sense) that we would like to hold. Constraints have been used for many years in interactive graphical applications for such things as specifying window and page layout [8]. They allow the designer to specify *what* are the desired properties of the system, rather than *how* these properties are to be maintained. The major advantage of using constraints is that they allow partial specification of the layout, which can be combined with other partial specifications in a predictable way. We note that constraints have previously been used for web-document layout [5, 2] and in animated Java applets [5].

In GXML/GXSL constraints are specified using a `<CONSTRAINT>` element which can occur within `GRAPH` elements. Among other properties, constraints can refer to the x and y location of elements within that graph as well as to global variables. For

[2] Our current prototype of a stand-alone GXML browser does not yet support full SVG tags. However, Section 4 suggest an alternative architecture which employs a standard SVG browser to display GXML graphs. This architecture significantly reduces the overhead for supporting full SVG in shape definitions.

example the constraint,

```
<CONSTRAINT TERM="pA.X >= ps.X + 2*ps.width"/>
```

constrains the node with ID `pA` to be to the right of the node with ID `ps` by at least 2 times the width of `ps`. This constraint is satisfied in both Figures 1 and 2. Clearly layout constraints can only be taken into account on elements that are actually displayed, e.g. if a constraint is defined on a node position within a subgraph and this subgraph is hidden or collapsed than this constraint can be disregarded by the layout module.

One complication in the use of constraints is that it is easy to accidentally define conflicting set of constraints. To allow for this we use the *constraint hierarchy* formalism [4]. A constraint hierarchy consists of a collection of constraints, each labeled with a strength. The relative strengths of the constraints give an order of preference if a decision has to be made which constraint to satisfy at the expense of another constraint violation.

There is a distinguished strength labeled `required`: Such constraints must be satisfied. There can be an arbitrary number of non-required strengths, and stronger constraints are satisfied in preference to ones with weaker strengths. The `required` label must be used judiciously, so as to ensure that the resulting constraint system is satisfiable. For this reason, by default constraints have the strength `strong` rather than `required`.

Given a system of constraints, it is the layout engine's task to find a solution to the variable attributes in the layout (positions, sizes) such that all required constraints are satisfied, and such that the non-required constraints are satisfied as well as possible in regard to the defined order of preference. For example, adding the weak constraint

```
<CONSTRAINT TERM="pA.X = ps.X" STRENGTH="weak"/>
```

which attempts to make the x coordinate of the two nodes equal, will have the effect of minimizing the distance between the x coordinates of the two nodes, since the previous constraint is stronger than this one.

The complete DTD for GXML is given in Appendix A.

3 GRAPH STYLE LANGUAGE (GXSL)

This section describes GXSL, the style definition language for GXML documents.

We could instead have used XSL, the generic style sheet language for XML. The major disadvantage of XSL is that it is complex and difficult to use, largely because it is a general purpose style language for all XML documents. As such, it does not support any special operations on graph structures, such as selection of connected nodes.

Another possibility would be to perform such operations by embedding Java Script methods into XSL, but this would complicate the language structure even further and would render such a style language unusable for the average web designer.

For these reasons we have designed a specialized style sheet language, GXSL, for specifying the appearance of GXML documents. GXSL is an XML-based language, just as XSL is defined in XML. Though this makes GXSL considerably more verbose than, for instance, CSS [1], it has the advantage of greater portability and offers a

```
<!DOCTYPE GXSL SYSTEM "GXSL.dtd">
<GXSL>
  <PRECONDITION>
    <CONSTRAINT TERM="Browser:frame-width >= 550px"/>
  </PRECONDITION>
  <LAYOUTER NAME="CSA1">
    <PARAMETER NAME="iterations" VALUE="50e3" />
  </LAYOUTER>
  <RULES>
    <RULE DESC="horizontally align all
                connected pairs of nodes">
      <LHS>
        <GRAPH>
          <NODE ID="_n1" Y="_y1"> </NODE>
          <NODE ID="_n2" Y="_y2"> </NODE>
          <WHERE>
            <EDGE FROM="_n1" TO="_n2"> </EDGE>
          </WHERE>
        </GRAPH>
      </LHS>
      <RHS>
        <CONSTRAINT TERM="(_y1=_y2)"/>
      </RHS>
    </RULE>
  </RULES>
</GXSL>
```

Figure 5: A simple GXSL file

number of possibilities for processing GXSL with standard XML tools, such as parsers, XSL processors etc.

A standard GXSL document can contain three main elements (see Figure 5). The first element <PRECONDITION> specifies preconditions that test whether a style sheet is applicable. The second element <LAYOUTER> specifies which layout engine is to be used to draw the graph. The third (and most interesting) element is a set of rules that are applied to the GXML file to derive additional layout conditions.

Preconditions are constraints which determine the applicability of the style sheet. They can only refer to various predefined read-only variables such as browser capabilities.

The layout engine must be specified, because each engine encapsulates a particular layout algorithm that determines the broad style in which the graph will be drawn, such as a straight-line graph, an orthogonal graph or a radial graph.

Though, at a first glance the rule structure appears very similar to the style of XSL rule files, there is a major difference: GXSL rules are not chained, i.e. each rule is applied in turn individually (and exhaustively), and rules cannot call other rules. This makes the semantics of GXSL documents easier to understand (and to define) than the more procedurally oriented semantics of XSL documents.

Each <RULE> consists of a left hand side <LHS> and a right hand side <RHS>. The left hand side defines a graph structure (using the same syntax as GXML) to be matched by an ordinary subgraph match in the GXML file on which the GXSL operates. If a match is found, the rule is applied. This means that the variables on the left hand side are bound according to this match and the right hand side is applied with appropriately instantiated variables. We call the left-hand side of the rule the *selector* and the right-hand side the *declaration*. The consequence of the rule application is that new layout constraints are added to the document.

Matching in GXSL is necessarily more powerful than in any other style sheet languages that we are aware of. The reason is that the elements of textual documents naturally form a hierarchy and so may be structured into a so-called *document tree*. For this case structure matching in style definitions depends mainly on parent-child or sibling relationships in the document tree. The whole structure of XSL is based on this idea. For graphs, however, this concept is not powerful enough: Though a GXML document, being an XML instance, is syntactically necessarily tree structured, the logical structure of the graph will usually not be tree-like. The style language must therefore allow for non-hierarchical and more complex structural matching.

For example, in our style sheet we might wish to horizontally align all of the children of a particular class of nodes, say MANAGER. In order to do this we need to match a node of class MANAGER using a variable (_par) and to match all its children, i.e., all nodes which share an edge with _par. We can then to horizontally align these children. The following GXSL rule does this:

```
<RULE>                                       (1)
  <VARIABLE ID="_childx">                    (2)
  <LHS>                                      (3)
    <NODE ID="_par" CLASS="MANAGER"></NODE>  (4)
    <GROUP ID="_children">                   (5)
      <NODE ID="_child"></NODE>              (6)
      <WHERE>                                (7)
        <EDGE FROM="_par" TO="_child"        (8)
              DIRECTED="TRUE"> </EDGE>       (9)
      </WHERE>                              (10)
    </GROUP>                                (11)
  </LHS>                                    (12)
  <RHS>                                     (13)
   <FORALL ID="_child" GROUPID="_children"> (14)
    <CONSTRAINT "_childx=_child.X"/>        (15)
   </FORALL>                                (16)
  </RHS>                                    (17)
</RULE>                                     (18)
```

Line (2) in the rule is a declaration for the variable _childx to which the x-position of all children of a node will be equated. This variable is local to the rule, and is essentially a new variable each time the rule is applied. Line (4) matches elements with tag <NODE> in the outermost graph element[3] whose CLASS attribute has value MANAGER. Note that the ID="_par" is a local name within the rule for <NODE>. It

[3]Note that there is an implicit top-level surrounding <GRAPH> element in the <LHS>.

does not ensure that the element being matched has an ID attribute with the value _par since an initial underscore (_) indicates that _par is a variable. The lines (5-11) collect the children of _par into a set of nodes identified by _children. This is similar to a list comprehension in functional languages. The remaining lines apply the layout constraint to each child in _children. Note that more complex alignment constraints can be defined on a higher-level using a special <ALIGNBOX> tag (see GXSL.DTD in Appendix B).

Since rule applicability is tested by subgraph matching, we need a method to explicitly restrict to total matches instead of subgraph matches. This is given by the <NOMORE/> tag. For example, the left hand side

```
<LHS>
   <GRAPH>
      <PORTLIST> <PORT ID="_a" ASSOC="_x">
                 <PORT ID="_b" ASSOC="_y">
      </PORTLIST>
      <NODE ID="_x"> </NODE>
      <NODE ID="_y"> </NODE>
      <EDGE FROM="_x" TO="_y"> </EDGE>
      <NOMORE/>
   </GRAPH>
</LHS>
```

will match all those subgraphs that contain precisely two nodes and one edge connecting them, but not any other elements, whereas the same definition without the NOMORE would match any subgraph which includes such a structure. More selective matching can be achieved by using the negation tag <NOT> in the WHERE clauses.

Some subtle problems arise, because matching, which can in principle be defined as subgraph matching on the underlying DOM structure, has to take hierarchical subgraph definitions into account. We provide two ways of matching: Simple matching as outlined above will match a subgraph only if it is explicitly marked as a subgraph in the left hand side pattern. However, more often than not this is not the intention: We want to be able to arbitrarily ignore subgraph boundaries when matching. This capability is provided by two mechanisms: (1) Bracketing a group of elements with an <ANY> tag results in a match that ignores subgraphs, i.e. if these elements occur at all in the graph, it does not matter at which level of the hierarchy they occur. (2) A second problem arises from the usage of ports as representatives of internal nodes in subgraphs. If, as above, we are looking to match, say, two nodes _x and _y connected by an edge no match would occur if _x and _y are on different levels in the subgraph hierarchy, because it is prohibited for an edge to connect them directly instead of using ports. To overcome this we have introduced a mechanism termed *port chasing*: An edge in a matching pattern can use two additional boolean attributes FROMPORTS and TOPORTS. If one of these attributes is true it will enable port chasing on the origin or destination side of the edge. This means that the match will ignore intermediate ports and will handle a multi edge that is routed through intermediate ports as if it connects its origin and destination directly.

For example consider the following rule that makes leaf nodes (those with no outgoing arcs) appear as circles.

```
<RULE>
  <LHS>
    <ANY>
      <NODE ID="_n1">
        <OBJSHAPE> <XMLVAR ID="_x"/> </OBJSHAPE>
      </NODE>
      <WHERE>
        <NOT>
          <EDGE FROM="_n1" TO="_n2"
                FROMPORTS="TRUE" TOPORTS="TRUE"
                DIRECTED="TRUE"> </EDGE>
        </NOT>
      </WHERE>
    </ANY>
  </LHS>
  <RHS>
    <STRUCTURE ID="_x">
      <CIRCLE-NODE/>
    </STRUCTURE>
  </RHS>
</RULE>
```

Applying this rule to the graph of Figure 3 will make the shape of nodes `pA`, `pB` and `enrol` circles.

The example makes use of another interesting feature of GXSL that helps to keep the document more compact: *structural DOM matching*. Recall that an OBJSHAPE can either be a primitive shape such as `<CIRCLE-NODE>`, or an arbitrary SVG element. In the example an XML variable `_x` is introduced that matches any XML structure (sub document tree) encapsulated as a child in the `<OBJSHAPE>` tag. On the right hand side of the rule we use the `<STRUCTURE>` tag to "explode" and instantiate the document subtree that was matched. In this case, the effect of the right hand side is to set the object shape for each node to a circle shape. Much more involved uses are possible, since the `<STRUCTURE>` tag actually uses unification. More precisely, the DOM tree bound to the XML variable during the matching of the `<LHS>` will be unified with the DOM tree corresponding to the child of the `<STRUCTURE>` tag.

To generalize edge matching, a `<PATH>` element is provided. It uses the same attributes `DIRECTED`, `FROM`, `TO`, `FROMPORTS` and `TOPORTS` as edges and has an additional attribute `LENGTH` that can be used to match the number of nodes on the path.

It should be noted that, unlike CSS, matching of the selection condition always starts at the root of the document tree: To implicitly match occurrences of elements at an arbitrary depth in the subgraph hierarchy one must enclose this element in an `<ANY>` tag as was done in the example. This instructs the matching algorithm to recursively descend into the subgraph structure (which effectively means to recursively descend into the DOM tree for the corresponding GXML) and to attempt the match at every level.

Finally, we note that it is possible to collect a set of all matching structures on the left hand side and iterate through the different matches on the right hand side applying the constraints to each element in this set. In the example this was done to collect a set of all children for every node and iterate through all the children using a local variable that does not change during this iteration. The tags <GROUP> and <FORALL> used for this capture the list comprehension nature of GXSL.

An obvious question is: How are style sheets associated with a GXML document? This is achieved by using the <STYLE> tag in the GXML document to "link" a style sheet with the document. On the client side, the viewer and browser may also implicitly link style sheets with the document. More than one style sheet may be linked either by multiple <STYLE> elements or by including a list of style sheets in a single element. If multiple style sheets are in the same <STYLE> element, the first applicable sheet is used (the others are ignored). If no style sheet's preconditions hold, none are imported. Consider the example directive

```
<STYLE hrefs="wide.gss,tall.gss,small.gss"/>
```

If `wide.gss`'s preconditions fail while `tall.gss`'s succeed, `tall.gss` is used. If, through the course of the user resizing the top-level browser frame, `wide.gss`'s preconditions later become satisfied, the layout does not switch to that style sheet unless `tall.gss`'s preconditions are no longer satisfied. That is, the choice among style sheets listed with one directive is only revisited when a currently-used style sheet is no longer applicable. We also note that in the case multiple layout engines are specified, the first will be used.

The complete DTD for GXSL is given in Appendix B.

4 SYSTEM

One of the most interesting issues regarding support for graphs on the internet is whether layout is performed on the client or server side or a combination of both. At least four possibilities exist:

- The simplest approach is to compute the layout once and for all on the server side. However, this is unsatisfactory because the layout cannot be flexibly changed to take into account client side requirements and capabilities. This almost leads back to the situation where a diagram is transmitted as a bitmap only.

- The second approach is a variant of this in which a new layout is computed on the server for each client—the client sends its requirements to the server and requests a layout for those requirements. This has the advantage that the server provides a dedicated layout engine which does not need to be downloaded but layout itself may still take some time.

- A third approach is to perform all layout on the client side: This certainly provides flexibility, but has the disadvantage of being comparatively slow and of requiring the client to download the layout engine (potentially slowing display). Of course, if we imagine a document format like GXML/GXSL as a standard,

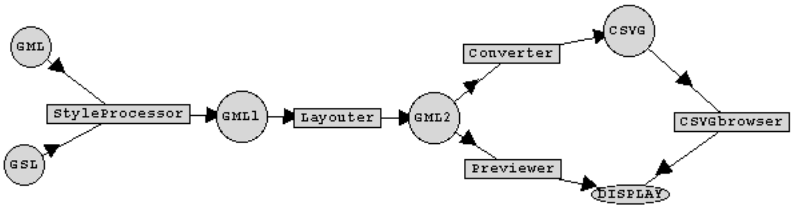

Figure 6: The GXML/GXSL system for graph layout.

appropriate browsers or plug-ins would also be a standard part of the client configuration, so that no additional download would be required. With further increasing processing power on the client side, a complete client layout will become realistic.

- The fourth approach is a combination of the first and second approach, in which an initial layout is computed on the server side, and then this layout is modified using simpler and fast techniques on the client side to take into account the client side requirements and capabilities. In this scenario, the layout modifier may only be able to handle a restricted class of requests on the client side.

We note that this division of layout responsibilities between client and server is similar to the different architectures discussed for constraint-solving for web document layout in [3].

In our prototype system we have elected to support the third and the fourth approach. The prototype system's architecture is shown in Figure 6. Recall that above we have discussed how GXML can contain "absolute" layout information, such as node coordinates, and "flexible" layout information, like requesting the placement of a label north-east of a node. As a consequence, GXML can be used as the *lingua franca* on all levels of the system architecture: To describe the logical structure without a layout, to give partial layout information, or to specify a precise layout completely.

The prototype system has four main components. The first is the *style preprocessor*. This takes a GXML document and its associated GXSL style sheets and applies the style sheets to the GXML document, giving rise to a new GXML document whose elements are annotated with style information. Since style sheets may have *preconditions* dictating when they are applicable, the preprocessor may generate multiple GXML layouts each with preconditions[4] determining when it is appropriate.

The second component is the *layouter*. This takes a GXML document and finds a layout for the graph described by the GXML document and, taking into account all (partial) layout information, produces a new GXML document whose elements are annotated with positioning information. Different graph styles may require different layouters as described above. The universal layouter that the prototype is using is described in Section 5.

[4] Although not previously mentioned preconditions are also allowed in GXML.

All these processes can be performed on the server side and/or on the client side. The delivery of the actual document to the client side can be done in one of two formats: Either a GXML document including precise layout information is sent to the client or it is converted into an SVG document [12] that is then delivered. In the first case a browser that is capable of displaying GXML is required, whereas in the second case, any standard browser with one of the widely used SVG plug-ins will do.

We are, in fact, not converting GXML to standard SVG but to an extended form of SVG, termed Constraint SVG (CSVG) [3]. This is an extension of SVG which supports linear arithmetic constraints and alternative layouts for groups of graphic elements. This extension has the same important implication that the integration of constraints into GXML has for graphs: By using constraint coordinates/sizes/etc. instead of absolute values, it becomes possible to let the client-side browser adapt the display automatically to the viewing conditions.

Consider the example of two nodes that have to be horizontally aligned. In GXML this could be expressed as

```
<NODE ID="node1">
  <LABEL> A node </LABEL>
</NODE>
<NODE ID="node2">
  <LABEL> Another node </LABEL>
</NODE>
<CONSTRAINT TERM="node1.Y=node2.Y">
```

This piece of GXML can be translated to a SVG fragment

```
<RECT ID="node1" X="100" Y="265" WIDTH="87" HEIGHT="15"/>
<TEXT ID="label1" X="100" Y="265" STYLE="font-size:10pt">
  A node
</TEXT>
<RECT ID="node2" X="200" Y="265" WIDTH="87" HEIGHT="15"/>
<TEXT ID="label2" X="200" Y="265" STYLE="font-size:10pt">
  Another node
</TEXT>
```

Note that the simple constraint of horizontal alignment cannot be passed on to the SVG code, since SVG requires an absolute coordinate. We can align the two nodes, but only in a particular position. This means that any interaction with the graph, such as moving one of the nodes would violate the constraint.

In order to better handle interaction and flexible layout for different viewing conditions, CSVG extends SVG by allowing constraints. The basic idea is that any attribute in SVG can be replaced by a variable with appropriate constraints attached. Using CSVG, the above GXML example code can be translated to the CSVG code:

```
<RECT ID="node1" X="100"  WIDTH="87" HEIGHT="15"/>
<TEXT ID="label1" X="100"  STYLE="font-size:10pt">
  A node
</TEXT>
```

```
<RECT ID="node2" X="200"  WIDTH="87" HEIGHT="15"/>
<TEXT ID="label2" X="200"  STYLE="font-size:10pt">
  Another node
</TEXT>

<CONSTRAINT ID="c1" TERM="node1.Y=node2.Y"/>
<CONSTRAINT ID="c2" TERM="node1.Y=label1.Y"/>
<CONSTRAINT ID="c3" TERM="node2.Y=label2.Y"/>
```

The CSVG viewer [3] maintains the relationships defined in these constraints automatically under changing viewing conditions. For example, if the user moves one node in the example, the other node will automatically stay aligned with it.

A current restriction of CSVG is that all constraints have to be linear, since for efficiency reasons the constraint solver in our CSVG prototype is a linear solver.

The layout engine and the GXML/CSVG converter therefore perform two essential roles. The layout engine generates an initial layout that fulfills the constraints in the GXML document and suggests concrete initial coordinates etc. for all elements. The CSVG converter "linearizes" the GXML document. This means that is filters the constraints in the GXML documents and only passes through the linear constraints to the CSVG document. It may also add new linear constraints to the document as a consequence of the layout process. Imagine the case where a special layout engine for directed acyclic graphs is used. Such an engine (for example using the Sugiyama algorithm [24]) would generate a layout like the one in Figure 2 even without providing additional constraints. The alignment visible in the example is simply a consequence of the layering induced by the graph's connection structure. However, when the layout is manipulated by the viewer, it is desirable to maintain the alignments that were generated by the layout engine. This is a case of constraints that are discovered and added by the CSVG converter. After the initial layout has been performed, the converter checks whether certain types of linear layout relationships (typically orthogonal ordering constraints) between the graph's elements occur in the concrete layout produced. If this is the case, it will add these as explicitly demanded constraints to the CSVG document so that they will be maintained by the browser.

The fourth component, the *previewer*, basically provides the same functionality, but without first converting the document to CSVG.[5]

The relationships of the components are illustrated in Figure 6.

In order to support the second suggested architecture in which all components reside on the client side, it would be possible to combine the style preprocessor, previewer and layouter we have implemented into a single client-side GXML layouter and renderer, which can be downloaded to implement by a browser so implementing the second architecture.

[5]Our current GXML previewer only provides limited user interaction.

5 LAYOUT ENGINE

As discussed in the introduction, the automated drawing of graphs is a computationally hard task and can be understood as a form of non-linear global optimization. Nevertheless, after more than two decades of intensive research into graph drawing algorithms, several different types of drawing methods have been developed which solve this problem for reasonably large graphs [10].

The ideal layout engine for our system should be widely applicable to many styles of graph layout and it must be able to take virtually arbitrary style constraints into account. Unfortunately, no current approach to graph drawing is sufficiently flexible to support this.

Many graph algorithms are highly specialized for particular types of graphs, such as orthogonal graphs, directed acyclic graph or radial trees. Such algorithms can achieve efficiency by applying problem-dependent special purpose heuristics or by tailoring a drawing algorithm narrowly to the particular problem. We can encapsulate any such algorithm into a special layout engine that a GXSL style sheet can use, but it is clear that its applicability would be very limited. Moreover, such algorithms are mostly unable to take into account additional user-defined constraints, such as alignment or preferred placement. In our context they would therefore be of limited use only. However, for some very common classes of graphs, such as trees, it would make sense to provide a specialized layout engine.

The second type of graph drawing methods are more general methods which can be applied to any graph. The most prominent example is the so-called Spring Embedder algorithm [11], which is based on a mapping of the layout problem to a mechanical problem: Each node in the graph is understood as a metal ring and the edges are understood as springs attached to these rings. The algorithm computes the minimum energy state for the spring system corresponding to the graph at hand. The position of the metal rings in the minimum energy arrangement corresponds to the position of the graph nodes in the final layout. Such generic approaches to graph layout provide a better basis for the kind of flexible layout that we require from our layout engine.

However, current generic approaches are not powerful enough to take into account virtually arbitrary style constraints. Certain types of additional layout constraints, such as minimum node sizes, orthogonality, directional information etc. can be encoded into some algorithms, such as in the Spring Embedder [25]. However, this is not very flexible since there is no generic technique for integrating new abstract mathematical layout constraints into this method: Each new kind of constraint requires manual engineering of a new type of spring embedder algorithm. The system described in [17] generalizes this approach in some sense: It uses an active set method to solve the spring embedding optimisation problem in the context of additional "abstract" constraints; however, the additional constraints are required to be linear.

The new generic approach we describe in this paper is flexible enough to take into account almost arbitrary (non-linear) constraints. In fact, as indicated earlier this method has been used to layout the figures in this paper from constraint-based GXML specifications.

Our starting point is the graph drawing algorithm of Davidson and Harel [9] which models graph drawing as a general non-linear numerical optimization problem and applies simulated annealing [21] to solve the corresponding numerical optimization problem. Simulated annealing is a powerful and popular global optimization procedure that can be used for a wide range of problems and the results of using it for graph layout are promising [16].

The idea is the following: We define a cost-function $f(g)$ that gives a numerical measure of "aesthetic desirability" to any graph. This cost function depends on a number of numerically measurable components $f_i(g)$ such as the uniformity of the node distribution, the average edge length, the standard deviation of the edge length, the number of edge crossings etc. All of these values are weighted and summed up, defining the overall desirability of a particular layout. In short, if G is the set of all possible layouts then we have a cost function f consisting of n components $f : G \to \mathbf{R}$ with $g \in G \mapsto \sum_{i=1\ldots n} w_i \cdot f_i(G)$.

We then apply a global optimization procedure to find the most desirable layout by minimizing the aesthetic cost function f. (The lower the value of the aesthetic cost function, the more desirable a layout is). Since this is a computationally hard global optimization problem, an adequate meta-heuristic must be used. Davidson and Harel have successfully applied simulated annealing to this problem, while other researchers have experimented with genetic algorithms [13]. Because of the more acceptable runtime behavior we adopt the simulated annealing approach.

Simulated annealing is a stochastic local search procedure used for global optimization that is based on the metaphor of annealing metals into the optimum molecular structure. The process starts from an initial configuration (here a random graph layout) and evaluates the cost function for this configuration. Next, it creates some new configuration by a random perturbation of the current configuration (here, moving some node along a random trajectory) and evaluates the cost function for this new configuration. If the new configuration is better than the previous one, it is accepted as the new current configuration. If it is inferior, it is not always rejected, but instead a probabilistic decision is made. The new configuration is accepted with a probability that depends on two parameters: The amount by which the new configuration has become worse and a virtual "temperature". In each iteration the temperature is decreased, so that it becomes increasingly unlikely with increasing runtime that moves for the worse are accepted. The whole algorithm is summarized in Figure 7.

As we are now using a universal non-linear optimization procedure to compute the layout, we can, in principle, incorporate arbitrary additional user-defined stylistic constraints into the cost-function $f(g)$. For each constraint we can simply measure if and how much this constraint is violated in a given configuration and add an appropriate "penalty" to the cost of this layout. Consider, for example, the simple case where we have k constraints $c_i(G)$. If the j-th constraint expresses the requirement that node a and node b be horizontally aligned then $c_j = \mid a.y - b.y \mid$. Our new evaluation function that takes these constraints into account is: $\tilde{f} : G \to \mathbf{R}$ with $g \in G \mapsto \sum_{i=1\ldots n} f_{i(G)} + \sum_{i=1\ldots k} p_i \cdot c_i(G)$.

If the amount of each penalty p_i is chosen correctly the normal simulated annealing procedure will produce layouts in which all constraints are satisfied and the cost function is at its optimum.

In practice, however, the choice of the penalties is the vulnerable element of the

```
Algorithm Drawing-SA
input: Graph G=(Vertices, Nodes);

   initialize layout Current := random_layout(G);
   initialize temperature T;

   while (T ≥ Tmin) do
      L := modification(Current);
      /* by moving some node with random direction and distance */
      if (cost(L) < cost(Current) then Current := L;
      else if ( random(0,1) < exp( −c/T × (cost(L)−cost(Current))) )
         then Current := L;
      else /* no change at all */
      decrease temperature T;
   end
```

Figure 7: Graph Drawing by Simulated Annealing

model. Penalties that are too low may be outweighed by the cost function and the algorithm may therefore produce layouts which do not satisfy the constraints. Too high penalties, on the other hand, prevent this but also reduce the relative importance of the cost function. In consequence, the algorithm will find layouts that satisfy the constraints, but may not be optimal in regard to the general aesthetic criteria.

A possible solution to this is to avoid fixing the penalties in advance by using dynamic penalties that change adaptively during the search. To the best of our knowledge, no such method has previously been applied to graph drawing problems. Many different kind of dynamic penalty schemas have been investigated for other types of problems, but these methods are generally ad-hoc and none of them performs really satisfactory over a wide range of problems [22]. However, recently a new extension of simulated annealing based on the theory of Lagrange multipliers in discrete spaces has been introduced in [26]. This method can be interpreted as a mathematically well-founded dynamic penalty method for simulated annealing.

It is this method that we apply to our problem of automatic graph drawing with user-defined constraints. The idea is simply to extend the graph drawing approach of Davidson and Harel with the general purpose constraint simulated annealing method presented in [26]. The basic difference to the original approach is that the search space which consists of all possible layouts in the case of standard simulated annealing, is extended by a new dimension which contains all possible penalty weights p_i. While the original algorithm only performs a probabilistic descent in the layout space, the new algorithm interleaves this with a probabilistic ascent in the penalty space. The whole algorithm is summarized in Figure 8.

In the same sense as basic simulated annealing is guaranteed to find the configuration with the optimum objective value, constraint simulated annealing is guaranteed to find the configuration in which all constraints are satisfied and the cost function is at its

```
Algorithm Drawing-CSA
input: Graph G=(Vertices, Nodes);

    initialize layout Current := random_layout(G);
    initialize penalties p[i] for i = 1 ... k;
    initialize temperature T;

    while (T ≥ Tmin) do
        if (random(0,1) > threshold) then
            /* modify layout */
            L := modification(Current);
            /* by moving some node with random direction and distance */
            if (cost(L) < cost(Current)) then Current := L;
            else if ( random(0,1) < exp(−c/T×(cost(L) − cost(Current))) )
                then Current := L;
            else /* no change at all */
        else /* modify penalties */
            i := random(1,k);
            if (ci(Current)=false) then
                /* ci(X) denotes whether the i-th constraint is satisfied by X */
                last := p[i];
                last_cost = cost(Current);
                modify p[i];
                /* by adding/subtracting a random value */
                if (cost(Current) > last_cost) then /* keep change */
                else if ( random(0,1) > exp(−c/T × last_cost −cost(Current)) )
                    then /* reset change */
                    p[i] := last;
                else /* no change at all*/
        decrease temperature T;
    end
```

Figure 8: Graph Drawing by Constraint Simulated Annealing

optimum.

At the same time, by using dynamic penalties, we are liberated from having to manually find appropriate penalties, which of course depend on the structure of the user-defined constraints. Thus this method can form a promising basis for general constraint graph drawing without additional user intervention.

It has to be noted that the flexibility of the general optimization approach has to be paid for with drawbacks in efficiency. Though Coleman and Parker [7] found efficiency improvements for the simulated annealing approach by exploiting information on the derivatives of the aesthetic function, this method cannot be used in our context as we are unable to make any assumptions on the constraints that will be specified.

The role of the simulated annealing approach is therefore that of the most general fallback method, for the case that no further information on the particular type of graph and/or the constraints is available. More specialized methods should be used if additional assumptions can be made. As a significantly faster alternative we have divised a constraint-based local search technique [15] that performs well on simpler types of graphs.

As constraint simulated annealing is computationally expensive, it is not suitable for supporting real-time interaction in a graph browser directly. As outlined above, our approach overcomes the problem of having to automatically maintain layout constraints in interactive editing mode by "linearizing" the style constraints once after the original layout computation. Thus only the resulting linear constraints have to be maintained during interactive editing. This can be performed efficiently with specialized solvers.

6 CONCLUSIONS

The web has become a medium that relies more and more on graphical communication. To this end we need better ways for communicating diagramatic information in a concise and flexible way which allows separation of layout and style from logical content and which allows the layout to take into account browser and viewer capabilities and requirements.

In this paper we have defined an approach to representing and displaying graph-based diagrams using an XML-based description language GXML. We allow the separation of graph structure descriptions from layout information by using the graph style language GXSL. Since automatic layout is a computationally expensive operation we have proposed a number of models for separating the layout and display of graphical information which divide work between client and server. We have built a prototype system for processing and laying out graphs described by GXML/GXSL. Laying out graphs in the GXML/GXSL framework is a challenging technical problem, and we have made an initial solution to this problem by constructing a general constraint graph layouter based on Constrained Simulated Annealing.

Although the work discussed in this paper focuses on graphs, it is clear that much of this discussion also applies to other forms of diagrams, such as statecharts or subway maps. The main reason why we are currently restricting the discussion to graph-based visualization is that relatively little is known about the automatic layout methods for more general types of diagrams. Generalizing to other diagrammatic notations will be an issue for future research.

References

[1] B. Bos et al. (eds.). Cascading Style Sheets, level 2 CSS2 Specification. Technical report, W3C, 1998. http://www.w3.org/TR/1998/REC-CSS2-19980512/.

[2] G. Badros, A. Borning, K. Marriott, and P.J. Stuckey. Constraint cascading style sheets for the web. In *Proceedings of the 1999 ACM Symposium on User Interface Software and Technology*, pages 73–82. ACM Press, 1999.

[3] G.J. Badros, J.J. Tirtowidjojo, K. Marriott, B. Meyer, W. Portnoy, and A. Borning. A constraint extension to scalable vector graphics. In *Tenth International World Wide Web Conference*, Hong Kong, May 2001.

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

[5] Alan Borning, Richard Lin, and Kim Marriott. Constraints for the web. In *Proceedings of 1997 ACM Multimedia Conference*, pages 173–182, 1997.

[6] T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler (eds.). Extensible Markup Language (XML) 1.0 (Second Edition). Technical report, W3C, 2000. http://www.w3.org/TR/2000/REC-xml-20001006.

[7] M.K. Coleman and D.S. Parker. Aesthetics-based graph layout for human consumption. *Software Practice and Experience*, 12(26):1415–1438, December 1996.

[8] I. Cruz, K. Marriott, and P. van Hentenryck (Eds.). Special issue on constraints, graphics and visualization. *Constraints*, 3(1), April 1998.

[9] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, October 1996.

[10] G. Di Battista, P. Eades, R. Tomassia, and I. Tollis. *Graph Drawing*. Prentice Hall, 1999.

[11] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[12] J. Ferraiolo (ed.). Scalable Vector Graphics (SVG) 1.0 Specification. Technical report, W3C, 2000. http://www.w3.org/TR/2000/CR-SVG-20001102/.

[13] A. Frick, C. Keskin, and V. Vogelmann. Integration of declarative approaches. In *Graph Drawing (GD'96)*, Berkeley/CA, September 1996.

[14] *International Symposium on Graph Drawing*. Springer-Verlag, Lecture Notes in Computer Science, 1994–2000.

[15] T. Hansen. *Constraint Graph Layout*. Honours thesis, Monash University, School of Computer Science, 2000.

[16] D. Harel and M. Sardas. Randomized graph drawing with heavy-duty preprocessing. *Journal of Visual Languages and Computing*, 6:233–253, 1995.

[17] W. He and K. Marriott. Constrained graph layout. In *Graph Drawing (GD'96)*, Berkeley/CA, September 1996. Springer.

[18] I. Hermann and M.S. Marshall. GraphXML, 1999. Reports of the Centre for Mathematics and Computer Science. http://www.cwi.nl/InfoVisu/GVF/GraphXML/GraphXML.pdf.

[19] M. Himsolt. GML—Graph Modelling Language. University of Passau, 1997. http://infosum.fmi.uni-passua.de/Graphlet/GML/.

[20] ISO/IEC JTC 1 and The VRML Consortium. The virtual reality modeling language. Technical report, ISO/IEC Standard 14772, 1997. http://www.vrml.org/Specifications/VRML97/.

[21] S. Kirkpatrick, C.D. Jr. Gelatt, and M.P. Vecci. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[22] Z. Michalewicz and D.B. Fogel. *How to Solve it: Modern Heuristics*. Springer-Verlag, 1998.

[23] S. Adler et al. (eds.). Extensible Stylesheet Language (XSL) Version 1.0. Technical report, W3C, 2000. http://www.w3.org/TR/2000/WD-xsl-20001018/.

[24] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2), 1981.

[25] R. Tamassia. Constraints in graph drawing algorithms. *Constraints, An International Journal*, 3:87–120, 1998.

[26] B.W. Wah and T. Wang. Simulated annealing with asymptotic convergence for nonlinear constrained global optimization. In *Proceedings of Principles and Practice of Constraint Programming*, pages 461–475. Springer-Verlag, 1999.

A The GXML DTD

The DTD for GXML is as follows:

```
<!ENTITY % SVG SYSTEM
   "HTTP://WWW.W3.ORG/GRAPHICS/SVG/SVG-19990730.DTD">
%SVG;

<!ELEMENT GXML (GRAPH*)>

<!ELEMENT GRAPH (LABEL*, OBJSHAPE?, PORTLIST?, SUBGRAPHS?,
                 CONSTRAINT*,GRAPH*, NODE*, EDGE*)>
   <!ATTLIST GRAPH ID ID #REQUIRED
                   DESC CDATA #IMPLIED
                   VISIBLE ( EXPANDED | COLLAPSED | OUTLINED
                             | HIDDEN ) "EXPANDED"
                   LAYOUT CDATA #IMPLIED
                   IMG CDATA #IMPLIED
```

```
                   ALT CDATA "">

<!ELEMENT LABEL (#PCDATA)>
   <!ATTLIST LABEL X CDATA #IMPLIED
                   Y CDATA #IMPLIED
                   PATHFRACTION CDATA "50"
                   CLASS CDATA ""
                   LABELPOS (N | NE | E | SE
                             |S | SW | W | NW ) #IMPLIED>

<!ELEMENT NODE (OBJSHAPE?, LABEL*)>
   <!ATTLIST NODE ID ID #REQUIRED
                  DESC CDATA #IMPLIED
                  CLASS CDATA ""
                  X CDATA #IMPLIED
                  Y CDATA #IMPLIED
                  VISIBLE ( TRUE | FALSE ) "TRUE">

<!ELEMENT EDGE (LABEL*, CONNECTOR?, ARROWHEAD?)>
   <!ATTLIST EDGE DIRECTED ( TRUE | FALSE ) "FALSE"
                  DESC CDATA #IMPLIED
                  CLASS CDATA ""
                  FROM IDREF #REQUIRED
                  TO IDREF #REQUIRED>

<!ELEMENT CONSTRAINT EMPTY>
   <!ATTLIST CONSTRAINT ID ID #REQUIRED
                        TERM CDATA #REQUIRED
                        STRENGTH ( REQUIRED | STRONG | WEAK ) "STRONG">

<!ELEMENT OBJSHAPE ((CIRCLE-NODE | TRIANGLE-NODE | RECTANGLE-NODE |
                     TEXT-NODE | ROUNDTANGLE-NODE)+ | SVG )>
   <!ATTLIST OBJSHAPE X CDATA #IMPLIED
                      Y CDATA #IMPLIED
                      WIDTH CDATA #IMPLIED
                      HEIGHT CDATA #IMPLIED>

<!ELEMENT CIRCLE-NODE EMPTY>
   <!ATTLIST CIRCLE-NODE X CDATA #IMPLIED
                         Y CDATA #IMPLIED
                         CX CDATA #IMPLIED
                         CY CDATA #IMPLIED
                         R CDATA #IMPLIED
                         FILLED ( TRUE | FALSE ) "FALSE">

<!ELEMENT TRIANGLE-NODE EMPTY>
   <!ATTLIST TRIANGLE-NODE X CDATA #IMPLIED
                           Y CDATA #IMPLIED
                           CX CDATA #IMPLIED
                           CY CDATA #IMPLIED
```

```
                           WIDTH CDATA #IMPLIED
                           HEIGHT CDATA #IMPLIED
                           FILLED ( TRUE | FALSE ) "FALSE">

<!ELEMENT ROUNDTANGLE-NODE EMPTY>
   <!ATTLIST ROUNDTANGLE-NODE X CDATA #IMPLIED
                              Y CDATA #IMPLIED
                              CX CDATA #IMPLIED
                              CY CDATA #IMPLIED
                              WIDTH CDATA #IMPLIED
                              HEIGHT CDATA #IMPLIED
                              FILLED ( TRUE | FALSE ) "FALSE">

<!ELEMENT RECTANGLE-NODE EMPTY>
   <!ATTLIST RECTANGLE-NODE X CDATA #IMPLIED
                            Y CDATA #IMPLIED
                            CX CDATA #IMPLIED
                            CY CDATA #IMPLIED
                            WIDTH CDATA #IMPLIED
                            HEIGHT CDATA #IMPLIED
                            FILLED ( TRUE | FALSE ) "FALSE">

<!ELEMENT TEXT-NODE EMPTY>
   <!ATTLIST TEXT-NODE STRING CDATA ""
                       LABEL CDATA "TIMES-ROMAN"
                       SIZE CDATA "10pt"
                       X CDATA #IMPLIED
                       Y CDATA #IMPLIED
                       CX CDATA #IMPLIED
                       CY CDATA #IMPLIED
                       WIDTH CDATA #IMPLIED
                       HEIGHT CDATA #IMPLIED
                       FILLED ( TRUE | FALSE ) "FALSE">

<!ELEMENT CONNECTOR (STRAIGHTLINE | CURVEDLINE | RECTLINE | SVG )>
   <!ATTLIST CONNECTOR XMIN CDATA #IMPLIED
                       YMIN CDATA #IMPLIED
                       WIDTH CDATA #IMPLIED
                       HEIGHT CDATA #IMPLIED
                       REFX0 CDATA #IMPLIED
                       REFY0 CDATA #IMPLIED
                       REFX1 CDATA #IMPLIED
                       REFY1 CDATA #IMPLIED>
                       <!-- REF determines the reference axis -->

<!ELEMENT STRAIGHTLINE EMPTY>

<!ELEMENT CURVEDLINE EMPTY>

<!ELEMENT RECTLINE EMPTY>
```

```
<!ELEMENT ARROWHEAD (SIMPLEARROW | SVG )>
   <!ATTLIST ARROWHEAD XMIN CDATA #IMPLIED
                       YMIN CDATA #IMPLIED
                       WIDTH CDATA #IMPLIED
                       HEIGHT CDATA #IMPLIED
                       REFX0 CDATA #IMPLIED
                       REFY0 CDATA #IMPLIED
                       REFX1 CDATA #IMPLIED
                       REFY1 CDATA #IMPLIED>
                       <!-- REF determines the reference axis -->

<!ELEMENT SIMPLEARROW EMPTY>

<!ELEMENT PORTLIST (PORT+)>

<!ELEMENT PORT (OBJSHAPE)>
   <!ATTLIST PORT ID ID #IMPLIED
                  ASSOC IDREF #REQUIRED
                  DESC CDATA #IMPLIED>

<!ELEMENT SUBGRAPHS (SUBGRAPH*)>
   <!-- FOR GROUPING COLLECTIONS OF BOXES INTO
        ARBITRARY NON-HIERARCHICAL BOXES -->

<!ELEMENT SUBGRAPH (NODEREF+, OBJSHAPE?)>

<!ELEMENT NODEREF EMPTY>
   <!ATTLIST NODEREF ID IDREF #REQUIRED>
```

B The GXSL DTD

The DTD for GXSL is as follows. Many of the definitions of the GXML DTD are required, most of the element definitions are extended to include the possibility of a matching variable. Unchanged !ELEMENT and !ATTLIST definitions from the GXML DTD are omitted.

```
<!ENTITY % SVG SYSTEM
   "HTTP://WWW.W3.ORG/GRAPHICS/SVG/SVG-19990730.DTD">
%SVG;

<!ELEMENT GSL (SOLVER, VARIABLES?, CONSTRAINTS?, RULES?)>

<!ELEMENT SOLVER (PARAMETER*)>
   <!ATTLIST SOLVER NAME CDATA #REQUIRED>

<!ELEMENT PARAMETER EMPTY>
   <!ATTLIST PARAMETER NAME CDATA #REQUIRED
```

```
                       VALUE CDATA #REQUIRED>

<!ELEMENT VARIABLES (VAR*)>

<!ELEMENT VAR EMPTY>
   <!ATTLIST VAR ID CDATA "_">

<!ELEMENT CONSTRAINTS (CONSTRAINT*)>

<!ELEMENT CONSTRAINT EMPTY>
   <!ATTLIST CONSTRAINT ID CDATA "_"
                        TERM CDATA #REQUIRED>

<!ELEMENT RULES (RULE*)>

<!ELEMENT RULE (VAR*, LHS, RHS)>
   <!ATTLIST RULE DESC CDATA "">


<!ELEMENT LHS (GRAPH)>

<!ELEMENT RHS (FORALL | STRUCTURE | CONSTRAINT | ALIGNBOX)+ >


<!ELEMENT FORALL (FORALL | STRUCTURE | CONSTRAINT | ALIGNBOX)+ >
   <!ATTLIST FORALL IDS CDATA #REQUIRED
                    GROUPIDS CDATA #REQUIRED >


<!ELEMENT STRUCTURE (GRAPH | LABEL | NODELIST | NODE | EDGELIST
                     | EDGE | OBJSHAPE | CIRCLE-NODE | TRIANGLE-NODE
                     | ROUNDTANGLE-NODE | RECTANGLE-NODE | TEXT-NODE
                     | CONNECTOR | STRAIGHTLINE | CURVEDLINE
                     | RECTLINE | ARROWHEAD | SIMPLEARROW | PORTLIST
                     | PORT | SUBGRAPHS | SUBGRAPH | NODEREF)* >
   <!ATTLIST STRUCTURE ID CDATA #REQUIRED>

<!ELEMENT ALIGNBOX EMPTY >
   <!ATTLIST ALIGNBOX ID CDATA "_"
                      GROUPID CDATA #IMPLIED
                      OBJECTS CDATA #IMPLIED
                      H-ALIGN (TOP | BOTTOM | CENTER | NONE) "NONE"
                      V-ALIGN (LEFT | RIGHT | CENTER | NONE) "NONE"
                      H-DISTRIBUTE (DIST | WIDTH | NONE) "NONE"
                      V-DISTRIBUTE (DIST | HEIGHT | NONE) "NONE"
                      X-DIST CDATA "_"
                      Y-DIST CDATA "_">

<!ELEMENT XMLVAR EMPTY>
   <!ATTLIST XMLVAR ID CDATA #REQUIRED>
```

```
<!ELEMENT GRAPH (XMLVAR |
                 ( (LABEL*, NOMORE?), OBJSHAPE?, PORTLIST?,
                   SUBGRAPHS?, CONSTRAINT*, (GRAPH*, NOMORE?),
                   (NODE*, NOMORE?), (EDGE*, NOMORE?), (PATH*, NOMORE?),
                   GROUP*, NOT?, ANY?, WHERE? ))>
  <!ATTLIST GRAPH ID CDATA "_" >

<!ELEMENT LABEL (#PCDATA | XMLVAR)*>

<!ELEMENT NODE (XMLVAR | (OBJSHAPE?, (LABEL*, NOMORE?)))>

<!ELEMENT EDGE (XMLVAR | ((LABEL*, NOMORE?), CONNECTOR?, ARROWHEAD?))>
  <!ATTLIST EDGE DIRECTED ( TRUE | FALSE ) "FALSE"
                   DESC CDATA #IMPLIED
                   CLASS CDATA ""
                   FROM IDREF #REQUIRED
                   TO IDREF #REQUIRED
                   FROMPORTS (TRUE | FALSE) "TRUE"
                   TOPORTS (TRUE | FALSE) "TRUE">

<!ELEMENT PATH EMPTY>
  <!ATTLIST PATH DIRECTED (TRUE | FALSE) "FALSE"
                 LENGTH CDATA #IMPLIED
                 FROM CDATA #REQUIRED
                 TO CDATA #REQUIRED
                 FROMPORTS (TRUE | FALSE) "TRUE"
                 TOPORTS (TRUE | FALSE) "TRUE" >

<!ELEMENT OBJSHAPE (XMLVAR |
                    ((CIRCLE-NODE | TRIANGLE-NODE | RECTANGLE-NODE
                      | ROUNDTANGLE-NODE | TEXT-NODE)+ | SVG))>

<!ELEMENT CONNECTOR (XMLVAR | (STRAIGHTLINE |
                              CURVEDLINE | RECTLINE | SVG ))>
   <!ATTLIST CONNECTOR XMIN CDATA #IMPLIED
                       YMIN CDATA #IMPLIED
                       WIDTH CDATA #IMPLIED
                       HEIGHT CDATA #IMPLIED
                       REFX0 CDATA #IMPLIED
                       REFY0 CDATA #IMPLIED
                       REFZ0 CDATA #IMPLIED
                       REFX1 CDATA #IMPLIED
                       REFY1 CDATA #IMPLIED>

<!ELEMENT ARROWHEAD (XMLVAR | (SIMPLEARROW | SVG ))>

<!ELEMENT PORTLIST (XMLVAR | (PORT+))>
```

```
<!ELEMENT PORT (XMLVAR | (OBJSHAPE))>

<!ELEMENT SUBGRAPHS (XMLVAR | (SUBGRAPH*))>

<!ELEMENT SUBGRAPH (XMLVAR | (NODEREF+, OBJSHAPE?))>


<!ELEMENT NOMORE EMPTY>

<!ELEMENT GROUP (VAR*,
                 (LABEL | GRAPH | NODE | EDGE | PATH
                  | GROUP | ANY | NOT )+, WHERE)>
   <!ATTLIST GROUP ID CDATA #REQUIRED
                   COLLECT CDATA #REQUIRED >

<!ELEMENT WHERE ((LABEL*, NOMORE?), OBJSHAPE?, PORTLIST?, SUBGRAPHS?,
                 (GRAPH*, NOMORE?),
                 (NODE*, NOMORE?), (EDGE*, NOMORE?), (PATH*, NOMORE?),
                 GROUP*, ANY?, NOT?, COND*)>

<!ELEMENT NOT ((LABEL*, NOMORE?), OBJSHAPE?, PORTLIST?, SUBGRAPHS?,
               (GRAPH*, NOMORE?),
               (NODE*, NOMORE?), (EDGE*, NOMORE?), (PATH*, NOMORE?),
               GROUP*, ANY?, NOT?, WHERE?)>

<!ELEMENT ANY ((LABEL*, NOMORE?), OBJSHAPE?, PORTLIST?, SUBGRAPHS?,
               (GRAPH*, NOMORE?),
               (NODE*, NOMORE?), (EDGE*, NOMORE?), (PATH*, NOMORE?),
               GROUP*, ANY?, NOT?, WHERE?)>

<!ELEMENT COND EMPTY>
   <!ATTLIST COND ID CDATA "_"
                  TERM CDATA #REQUIRED>
```