GraphXML — An XML based graph interchange format

I. Herman, M. S. Marshall
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Email: {I.Herman, M.S.Marshall}@cwi.nl

ABSTRACT

GraphXML is a graph description language in XML that can be used as an interchange format for graph drawing and visualization packages. The generality and rich features of XML make it possible to define an interchange format that not only supports the pure, mathematical description of a graph, but also the needs of information visualization applications that use graph–based data structures.

1998 Computing Reviews Classification System: D.2.12, H.3.5, I.3.6, I.3.8, I.7.2,
Keywords and Phrases: information visualization, graph visualization, user interfaces, XML
Note: The work was carried out under the project INS3.1 "Information Visualization".

1 INTRODUCTION

GraphXML is a graph description language in XML*. The goal of GraphXML is to provide a general interchange format for graph drawing and visualization systems, and to connect those systems to other applications. The requirements of information visualization have greatly influenced design decisions during the development of GraphXML. Although GraphXML can be used for the description of purely mathematical graphs, restricted to the set of nodes and edges, information visualization applications require more features. For example, it should be possible to label graphs, nodes, and edges. It should also be possible to attach application-dependent data and external references. Applications might produce not just a single graph, but also a whole series of graphs, possibly ordered in time. We might want to control the visual appearance of the graphs, such as the colour of the edges, images used when iconifying a window containing a specific graph. The interchange format should be able to cope with these demands as well.

As part of a larger project in graph visualization, we needed a graph description language. We initially looked for an existing standard or widely used format. We also hoped to find an existing parser in Java for such a graph description, which would save us from having to develop our own. However, such a format does not exist. The closest we found is a language called GML[4], which was originally proposed as a general interchange format in the graph drawing community. Although GML is a capable description language for graph drawing purposes and includes provisions for extension, the mechanism for associating external data with a graph element is not well-defined, which would lead to a proliferation of incompatible GML dialects when adapted to specific applications. Other formats exist, such as WebDot's DOT format[5], but also couldn't support the needs of information visualization. Consequently, we decided to develop our own format, in an effort to meet those needs. We based the format on XML in order to take advantage of several features and the benefits of an existing standard. The detailed specification of our file format, as well as the Java–based parser we have developed, have been put into the public domain. We hope that the format will enter widespread use. In our view, this would be beneficial for the graph visualization community.

To use the terms defined for XML[1,2,3] , GraphXML is an "application"† of XML. There are several reasons for choosing XML as the basis, which included:

- XML is used by many different application areas to define input and output formats including those for databases, chemical compound definitions, and schematic graphics. A graph interchange format based on XML has a greater chance of being accepted by other application communities.

* XML is a specification developed by the World Wide Web Consortium (http://www.w3c.org).

† The term XML "vocabulary" is also used.

- XML defines clear syntactic rules on how to define a specific "language" for an application through the Document Type Definition (DTD) files. These rules allow extensions as well. What this means is that, although a precise specification of GraphXML is provided, the end–user has the option of adding his/her extensions to GraphXML, if clear guidelines are followed.
- There are a number of XML–based specifications that are being defined by communities, both within and outside the World Wide Web Consortium. GraphXML can reuse some of these applications. An example is XLink[4], which defines rules for linking to other information, much like the `<a>` tag of HTML. Another example is a specification of a binary exchange format for XML documents: although it is still in a planning phase, such a format can significantly speed up the transfer and the processing of these files.
- A large number of software tools are emerging, which are either based on XML or work with XML. For example, the new releases of the Web browsers (Internet Explorer 5, the upcoming Netscape 5 and Opera Browser Version 4) can, or will be able to, view XML files. There are also a number of XML editors, both as products and in the public domain. Although these applications do not replace graph visualization tools, they can be very helpful in managing graph description files, and they can facilitate the development of new visualization tools and their interfaces with other applications.
- A number of XML parsers are available, some of them free of charge. Of course, these packages cannot be used directly for an application like GraphXML: the semantic interpretation of the application is still to be provided. Nevertheless, these packages form the bulk of a full parser, including error management, syntax checking, etc. In our own experience, developing a parser for an application like GraphXML becomes a relatively straightforward work, compared to what would be needed to develop a full parser even if tools like yacc or bison were used. This development is greatly facilitated by the fact that the World Wide Web Consortium has also defined a Document Object Model (DOM)[6], i.e. a set of class specification representing the run–time "view" of the XML parse tree. This DOM model is implemented by most packages. As a consequence, a specific parser, such as the one we have developed for GraphXML, can be implemented in such a way that the dependency on a particular XML can be greatly reduced.

In what follows, an overview of the main features of GraphXML is given. This is followed by a more detailed specification of all tags used in GraphXML (Section 7). A rudimentary knowledge of XML, or at least of HTML, is required to understand this description.

2 GRAPH STRUCTURES IN GRAPHXML

2.1 A simple example

The following code segment shows the simplest possible use of GraphXML that describes a graph with two nodes and a simple edge:

```
1    <?xml version="1.0"?>
2    <!DOCTYPE GraphXML SYSTEM "file:GraphXML.dtd">
3    <GraphXML>
4      <graph>
5        <node name="first"/>
6        <node name="second"/>
7        <edge source="first" target="second"/>
8      </graph>
9    </GraphXML>
```

This example shows the basic style of a graph description in GraphXML. It greatly resembles the way HTML documents are written, albeit using different tags.

The first line is required in all XML files. The second line identifies the file's type, i.e. that this is an XML application based on the document type description called `GraphXML.dtd`*. Finally, the third and the last lines enclose the real content of the files, much like the `<html>` tag that precedes and closes a well–formed HTML file. The real

* To be precise, this line specifies that the DTD file is on the local file system. This could be replaced by a URL specification, placing the DTD file onto the World Wide Web for public access.

content begins with line number 4, which defines a full graph. We delineate graph definitions with the <graph> tag so that a file can contain several graph definitions. The body of the graph description is quite straightforward: two nodes and a connecting edge are defined. Note that adding the <node> tag is not compulsory; the semantics of the parser is such that if an edge is defined with nodes, and the nodes are not (yet) specified, a "minimal" node will be created on the fly. If no additional attributes are attached to the nodes, this can lead to a significant saving in file size for large graphs.

Much like HTML, "attributes" can also be defined for each of the elements: these are key–value pairs. The document type definition of GraphXML defines, for each element, the set of allowable attributes, and the parser is supposed to check those. It is partly through those attributes that additional information about nodes, edges, or graphs can be conveyed to the application. For example, the <graph> tag can use keys such as version, vendor, preferredLayout, isPlanar, isDirected, isAcyclic, or isForest (the first two attributes can have any string value, the third one can refer to a series of well accepted "names" for layout algorithms, the others can only take on the values of true or false). The semantics attached to these keys is self-explanatory*. For example, the graph specification above could be:

```
4     <graph isPlanar="false" isDirected="true" vendor="CWI, Amsterdam" version="1.0">
```

The example above contains all the elements that are necessary to describe a purely mathematical graph, without any connection to an external application. However, in applications, graphs usually represent domain-specific data along with its semantics, which must also be represented in the interchange format.

2.2 Application dependent data

Application data can be added to different levels of the graph description through a series of additional elements defined by GraphXML. These elements are meant to represent the different types of data that might be assocaited with a graph, a node, or an edge. GraphXML defines the following application data:

- **Labels** (<label> tag): whereas the name attribute is used for identification, which means that it must be unique within one graph description, the label can contain any kind of text and does not have to be unique. Applications can use these to label nodes and edges, or as a title in the window.
- **Data** (<data> tag): application-dependent data represented by a node, edge, or even the full graph. The <data> tag can contain any kind of information that can be described in XML†.
- **Data references** (<dataref> tag): application dependent data, which, instead of being directly incorporated into the graph files, is referred to through external references.

The format of external references (within the <dataref> tag) follows the specification of a separate document of the World Wide Web consortium, called Xlink[7]. For our purposes it is sufficient to say that the format of the references is virtually identical to the URL formats used in HTML.

The following example describes the same graph as in the previous section, except that the first node is enriched with application dependent data. The kind of data used in the example may be appropriate for the input of a visual Web Navigator: the data attached to the first node gives a more detailed description of the represented Web page.

```
1     <?xml version="1.0"?>
2     <!DOCTYPE GraphXML SYSTEM "file:GraphXML.dtd">
3     <GraphXML>
4       <graph>
5         <node name="first">
6           <label>Project Home page</label>
7           <data>
8             This is a description of the CWI Information Visualization project.
9           </data>
10          <dataref>
```

* Some of these attributes have a default value, which is automatically added to the result of the parsing if the user does not provide one. This is an example of a small, but very helpful feature provided by XML and the parser APIs cited above.

† Although XML describes everything in terms of strings, it has its own formalism, called entities and notations, which can be used to include binary data, too. However, using external references, through the <dataref> tag, may be more appropriate for this.

```
11            <ref xlink:role="Project leader" xlink:href="http://www.cwi.nl/~ivan"/>
12            <ref xlink:role="Description" xlink:href="http://www.cwi.nl/InfoVisu"/>
13          </dataref>
14        </node>
15        <node name="second"/>
16        <edge source="first" target="second"/>
17      </graph>
18    </GraphXML>
```

Note the use of the `xlink:role` attribute in the example (see lines 11 and 12), which can be used to describe what the exact role of the link is. This can provide a useful indication to the application.

It is important to note that XML files form, structurally, a tree: a <dataref> tag is the child of the containing <edge> or <node> tag, these are children of <graph>, etc. This simple observation is important to understand the exact details of the semantics attached to the elements; example will follow in the forthcoming sections.

2.3 Hierarchical graphs

All the examples mentioned until now refer to one graph. However, information visualization applications usually use very large graphs, and one of the ways of handling this problem is to define the information in terms of a hierarchy of graphs, or clusters, instead of one single graph. What this means is that the nodes of one graph can refer to other graphs, these can refer to yet another graph, and so on. Powerful techniques exist to cluster graphs into hierarchies and to visualize the hierarchies (see [8,9] or the survey of Herman *et. al*[10] for further details).

GraphXML offers a way to describe such graph hierarchies. Consider the following example:

```
19    <?xml version="1.0"?>
20    <!DOCTYPE GraphXML SYSTEM "file:GraphXML.dtd">
21    <GraphXML>
22      <graph id="levelOne-1">
23        <node name="first"/>
24        <node name="second"/>
25        <edge source="first" target="second"/>
26      </graph>
27
28      <graph id="levelOne-2">
29        <node name="third"/>
30        <node name="fourth"/>
31        <edge source="third " target="fourth"/>
32      </graph>
33
34      <graph id="levelTwo">
35        <node isMetanode="true" name="cluster1" xlink:href="#levelOne-1"/>
36        <node isMetanode="true" name="cluster2" xlink:href="#levelOne-2"/>
37        <edge source="cluster1" target="cluster2"/>
38      </graph>
39    </GraphXML>
```

Although very simple, the example shows the tools introduced in GraphXML to describe graph hierarchies. First, a single GraphXML document can contain several graph descriptions. Each graph description can use a (unique) identifier, using the `id` key. A “cluster” or “meta” node in a higher level in the hierarchy uses this identifier to “`link`” to another graph, using the `xlink:href` attribute key (see line numbers 35 and 36). The `isMetanode` attribute is used to unambiguously identify a node that refers to another graph. Using these definitions, this example describes a two level graph with two nodes and one connecting edge, where each node represents another graph.

The `xlink:href` attribute for a meta node follows the same rules as the attribute in the <a> tag of HTML when used to identify a target within a document. The full format of the value is:

```
URL#identifier
```

where the identifier refers to a graph identifier *within* the document referred to by the URL. If the target is in the same document as the source, the URL part can be left out. This version is used in the example.

This simple adaptation of the well–known rules of HTML introduces an extremely powerful feature to GraphXML. It is indeed possible to define a clustering hierarchy for a graph that is in another file, possibly on another Internet location than the cluster description itself. Applications making use of this possibility can use their own clustering techniques to visualise public datasets.*

2.4 Dynamic graphs

If a graph visualization system is used interactively, the system may be asked to store the history of the user's actions in some form of journaling. What this means is that an interchange format should be able to describe not only the initial graph, but also any editing steps that have changed the structure or the attributes of the graph. This is one example when the editing tags of GraphXML become essential.

The edit sections in an GraphXML are syntactically similar to graph specification, except for the use of the `<edit>` tag instead of `<graph>`. Furthermore, the `<edit>` tag has a required attribute keyed `action`, whose value can be `remove` or `replace`. Here is an example:

```
1    <?xml version="1.0"?>
2    <!DOCTYPE GraphXML SYSTEM "file:GraphXML.dtd">
3    <GraphXML>
4      <graph version="1.0" vendor="cwi" id="theGraph">
5        <node name="first">
6          <label>A label on this node</label>
7          <dataref>
8            <ref xlink:href="BigIcon.gif"/>
9            <ref xlink:href="MediumIcon.gif"/>
10           <ref xlink:href="SmallIcon.gif"/>
11         </dataref>
12       </node>
13       <node name="second"/>
14       <edge name="thisEdge" source="first" target="second">
15         <dataref>
16           <ref xlink:href="ExternalData.bmp"/>
17         </dataref>
18       </edge>
19     </graph>
20
21     <edit action="replace" xlink:href="#theGraph">
22       <node name="first">
23         <label>Another label</label>
24         <dataref>
25           <ref xlink:href="anotherImage.gif"/>
26           <ref xlink:href="tiny.gif"/>
27         </dataref>
28       </node>
29       <edge name="thisEdge" source="first" target="second"/>
30     </edit>
31   </GraphXML>
```

The semantics of the editing element (line 21) is as follows: in general, each element in the XML hierarchy is defined by its position in the hierarchy; an icon group belongs to a specific node, the latter to a graph, and so on. In principle, all elements have all their possible sub–elements defined, by possibly empty values. In other words, if no label has been defined for a node, it is considered to be defined with an empty label. To perform editing, the application has to find the "deepest" element in the hierarchy in the specification in the edit element, *not including the links within the* `<datarefs>` *tag* (i.e. it is not possible to individually edit the content of this "collection"-like entity). This controls what is being edited. The value of the `action` attribute determines what happens: the corresponding element will either be removed (i.e. replaced by an empty element) or will be replaced by the content in the `<edit>`. The restriction on the data references means that it is not possible to edit the detailed content of these elements, only as complete units.

* Note that WebDot's DOT format includes facilities to describe clusters, but the format is limited to local graphs only.

The result of the editing action in the example above can be represented by an GraphXML description as follows:

```
1     <?xml version="1.0"?>
2     <!DOCTYPE GraphXML SYSTEM "file:GraphXML.dtd">
3     <GraphXML>
4       <graph version="1.0" vendor="cwi" id="theGraph">
5         <node name="first">
6           <label> Another label </label>
7           <dataref>
8             <link xlink:href="anotherImage.gif"/>
9             <link xlink:href="tiny.gif"/>
10          </dataref>
11        </node>
12        <node name="second"/>
13        <edge name="thisEdge" source="first" target="second"/>
14      </graph>
15    </GraphXML>
```

Note the disappearance of the data references in the edge (lines 15–17 in the previous example). This is because the editing action has replaced those with their “empty” counterpart from within the editing element (line 29 in the previous example).

If, in the same example, the `action` attribute was set to `remove`, the result would be as follows:

```
1     <?xml version="1.0"?>
2     <!DOCTYPE GraphXML SYSTEM "file:GraphXML.dtd">
3     <GraphXML>
4       <graph version="1.0" vendor="cwi" id="theGraph">
5         <node name="first">
6         </node>
7         <node name="second"/>
8         <edge name="thisEdge" source="first" target="second">
9         </edge>
10      </graph>
11    </GraphXML>
```

The `xlink:href` attribute used by the edit element has the same syntax and semantics as described for hierarchical graph descriptions. In other words, an edit element can refer to a graph in another file or even another Internet location, too.

As a final touch to editing, GraphXML also introduces the `<edit-bundle>` tag, which is simply a sequence of edit tags:

```
1     <edit-bundle>
2       <edit …> … </edit>
3       <edit …> … </edit>
4     </edit-bundle>
```

This simple grouping of editing elements can be useful if the application wants, for example, to animate the result of editing, but it is unnecessary to do this on the level of granularity of the individual editing steps. Using this bundling mechanism, animation can be controlled by the creator of the GraphXML file.

3 STORING GEOMETRY

All the examples in Section 2 described only structural elements: nodes, edges, and hierarchies. Visualization systems have to layout the graph before presenting it to the user. Layout algorithms can be complex and time consuming. Therefore, it is advantageous if the graph description can store the geometric positions of the nodes and the edges. If this information is available, the visualization system might decide to use those instead of calculating new layout positions, thereby saving a significant amount of time. Furthermore, the graph description can become a real interchange file between different visualization systems, preserving not only structure but also geometry.

3.1 Node positions

The position of a node can be described by adding the <position> tag as a child to <node>. Syntactically, the element itself is simple:

```
1    <position x="0.0" y="0.0" z="0.0"/>
```

which, in this case, assigns the node to the origin in 3D space. Any of the three coordinates can be omitted, in which case their default value is 0.0. Especially, the user can choose to ignore the z coordinates in all positioning elements in a graph, thereby placing the full graph simply onto 2D. As a convention, the coordinate system is right–handed, which also means that, in 2D, the origin is in the lower left hand corner.

Using the <size> tag, the size of the node can also be described:

```
1    <size width="3.0" height="5.0" depth="4.0"/>
```

In contrast the position, the width and height attributes are required if this element is used. The depth attribute can be omitted, in which case its value is set to 0.0 by default. This tag might be especially important for some layout algorithms, which may want to take the node size into account when laying out the graph; this means that this tag might be used in isolation, too, without using the <position> tag.

3.2 Edge positions

Edges differ from nodes insofar as a sequence of coordinates may be necessary to describe a particular polyline or a curve for the edge. This is achieved through the <path> tag:

```
1    <path type="polyline">
2        <position x="0.0" y="0.0"/>
3        <position x="0.1" y="0.0"/>
4        <position x="0.1" y="0.1"/>
5    </path>
```

which contains a sequence of control points. The type attribute can take the value of polyline, arc, or spline, depending on whether the edge is to be drawn as a polyline or a spline curve. In the case of a spline, the positions indicate the spline control points.

3.3 Graph size

The <size> tag can also be used as a direct child element of <graph>. It then denotes the full size, or bounding box, of the full graph. Applications can greatly benefit from such information, because they can allocate the right area on the screen and set up the necessary coordinate transformations in advance. Using all the elements defined so far, here is how geometry information could be added to the simple example of page 2:

```
1    <?xml version="1.0"?>
2    <!DOCTYPE GraphXML SYSTEM "file:GraphXML.dtd">
3    <GraphXML>
4      <graph id="basic">
5        <size width="1.2" height="1.2"/>
6        <node name="first">
7            <position x="0.0" y="0.0"/>
8            <size width="0.1" height="0.1"/>
9        </node>
10       <node name="second">
11           <position x="1.0" y="1.0"/>
12           <size width="0.2" height="0.2"/>
13       <edge source="first" target="second">
14           <path type="polyline">
15               <position x="0.0" y="0.0"/>
16               <position x="0.1" y="0.0"/>
17               <position x="0.1" y="0.1"/>
18           </path>
19       </edge>
20     </graph>
21   </GraphXML>
```

3.4 Geometry for hierarchical graphs

The geometry definition described in the earlier section become underspecified if hierarchical graphs are used: the same graph can be included at various places in the second order graph, the geometry must be adapted to the metanode's position, etc. The solution is to use the <transform> tag, which is a child of <node>. This element has no meaning if the node is not a metanode. Otherwise, it describes the transformation that has to be applied on each coordinate value in the referred graph. The transformation element contains the specification of a transformation matrix. To take a specific example, the following fragment:

```
1        <node name="SecondOrder" isMetanode="true" xlink:href="#basic">
2            <transform matrix="1.0 0.0 0.5 0.0 1.0 0.5"/>
3        </node>
```

refers to our example above, but translates all the nodes and points to the (0.5,0.5) point.

The <transform> element contains either 6 numbers to describe a 2×3 matrix row–by–row, or 12 numbers to describe a 3×4 matrix, again row–by–row. In both cases, the convention is that coordinate vectors are column vectors, multiplied by the transformation matrix from the left (see, for example, [12] for further details on how these transformation matrices are used in computer graphics).

4 VISUAL PROPERTIES

Beyond the pure geometry, the visual appearance of a graph is also determined by visual properties, such as line width, colour of the components, icons replacing nodes, etc. It is a natural requirement to store those values, too.

In GraphXML, the simplest way of controlling these properties is to use the <style> tag as a direct descendent of a node or an edge. A style can include the tags <line> or <fill>. In the case of a node, the line tag controls the border of the symbol drawn for the node, whereas the fill tag controls the interior. For example, the fragment:

```
1    <node name="first">
2        <style>
3            <line linestyle="dashed" linewidth="1.2" colour="red"/>
4            <fill fillstyle="solid" colour="blue”/>
5        </style>
6    </node>
7    <edge source="first" target="second">
8        <style>
9            <line linestyle="solid" linewidth="1.0" colour="cyan"/>
10           <fill fillstyle="none"/>
11       </style>
12   </edge>
```

defines a node symbol to have a red dashed boundary drawn with a line width of 1.2, and filled with solid blue*. The edge is to be drawn in cyan without filling the interiors (in the case the edge is drawn as a polygon, for example). There are 13 predefined colours that can be referred to through their symbolic names; alternatively, the binary value for the RGB or RGBA values can be specified directly†. Similarly, there are four predefined line styles, but a 16 bit value integer can also be specified which acts as a dash pattern for the line.

The fill element can also refer to an image file instead of specifying the colour and the fill style. This instructs the visualizer to use the image as an icon to display the node. For example, line 4 could be replaced by:

```
4            <fill xlink:href="http://www.some.site/imagefile.gif"/>
```

to use an icon for the node.

Although the mechanism described so far is useful, it would lead to many repeated visual control tags, greatly increasing the size of the graph file. In addition, it might become cumbersome to adapt a graph file to a new envi-

* Note that this specification does not specify the exact glyph to be drawn by the visualizer. This is either left to the implementer of the visualization system, or specified via a more sophisticated control tag, called <implementation>. See Section 7.9.3 for further details.

† For those who prefer the American spelling to British, the attribute name "color" is also accepted.

ronment with other visual characteristics. To solve these problems, we added a more complex mechanism to the visual property control. It is inspired by the general principle of separating style from content: whereas the structural content of a graph, as well as its geometry, is an inherent part of the graph description, visual properties may change from one environment to the other. The goal is to provide a general mechanism that allows for an easy adaptation.

First of all, a style element can be added on the graph level, to control the overall apperance of the graph. This means that the example above could be replaced by:

```
1    <graph>
2        <style>
3            <line tag="node" linestyle="dashed" linewidth="1.2" colour="red"/>
4            <fill tag="node" fillstyle="solid" colour="blue"/>
5            <line tag="edge" linestyle="solid" linewidth="1.0" colour="cyan"/>
6            <fill tag="node" fillstyle="none"/>
7        </style>
8        …
9        <node name="first"/>
10       <edge source="first" target="second"/>
11       …
12   </graph>
```

resulting in the same visual effect, except that the visual properties are valid for *all* nodes and edges in the graph (note the use of the `tag` attribute in the line and fill elements to differentiate between nodes and edges). There is an exception, though: if a node does have a local style specification for one of the visual properties, this takes priority over the global value.

To make one step further, nodes and edges can also use the `class` attribute to categorize tags with common visual properties. Using this additional identification (which is different from the `name`, because it is not necessarily unique for an element), a finer control over the visual attributes can be achieved by targeting a specific visual attribute to a class of nodes or edges. For example, in

```
1    <graph>
2        <style>
3            <line tag="node" linestyle="dashed" linewidth="1.2" colour="red"/>
4            <fill tag="node" fillstyle="solid" colour="blue"/>
5            <fill tag="node" fillstyle="solid" colour="green" class="special"/>
6            <line tag="edge" linestyle="solid" linewidth="1.0" colour="cyan"/>
7            <fill tag="node" fillstyle="none"/>
8        </style>
9        …
10       <node name="first"/>
11       <node name="second" class="special"/>
12       …
13       <node name="nth" class="special"/>
14       …
15       <edge source="first" target="second"/>
16       …
17   </graph>
```

the node `first` will be displayed the same way as before; however the nodes `second` and `nth` will become green instead of red. This is because line 5 specifies that “all nodes of class ‘special’ should be filled in green”, which is the case for `second` and `nth`.

In general, the class attribute value is set for each element by the author of the XML file. There is one exception, though: all metanodes are automatically assigned the `class="metanode"` attribute (unless an explicit class assignment overrides this).

The `<style>` tag can also appear as a direct child element of the root (i.e. `<GraphXML>`) affecting all graph specifications in the file. Going from a leaf (i.e. a node or an edge) in the XML tree toward the root, the priority of the visual property statements diminish: if a property is specified in the graph, this overrides the specification on the root level, the element level description overrides the graph level one, etc. Finally, the class related specifications override the general specifications.

Another syle control facility is special to metanodes. Consider the following example:

```
1     <?xml version="1.0"?>
2     <!DOCTYPE GraphXML SYSTEM "file:GraphXML.dtd">
3     <GraphXML>
4       <graph id="levelTwo">
5         <node isMetanode="true" name="cluster1" xlink:href="http://www.someurl">
6           <subgraph-style>
7             <line tag="edge" colour="red"/>
8           </subgraph-style>
9         </node>
10        …
11      </graph>
12    </GraphXML>
```

The graph refers, through a metanode, to another graph which, in this case, happens to be on a remote Internet location. By adding the `<subgraph-style>` element, this specification will control the visual apperance of the included graph. Semantically, this is similar to a `<style>` element placed on the file level in the remote graph. However, the user is not forced to modify the remote graph in order to gain control over the visual properties. As with the priority of other style statements, these visual properties have a lower priority than the statements in the remote file.

Using the entity mechanism of XML, it is possible to include an XML file in another (see Section 9 on how this can be done). This is particularly handy when controlling styles: it is possible to collect all the style elements into a separate file and include it in a graph specification. If a change is done in that style file, all graphs using it will have their visual properties automatically updated.

5 USER EXTENSIONS

Although the specification of GraphXML includes rich facilities for the addition of data to node and edge definitions, it is not possible to predict all possible information an application might want to add. For example, if the application is a web visualizer, it might want to add information on a MIME type for a node. In other words, the application would need to have its own, `<mime>` tag for each node, beyond those defined by the basic GraphXML.

Such extensions are possible using GraphXML. This is how an extension mechanism can be added to a graph:

```
13    <?xml version="1.0"?>
14    <!DOCTYPE GraphXML SYSTEM "file:GraphXML.dtd" [
15    <!ENTITY % nodeExtensions "|mime">
16    <!ELEMENT mime EMPTY>
17    <!ATTLIST mime
18        type        CDATA #REQUIRED
19        application CDATA #IMPLIED
20    >
21    ]>
22    <GraphXML>
23      <graph>
24        <node name="first">
25          <label>Project Home page</label>
26          <dataref>
27            <ref xlink:href="http://www.cwi.nl/InfoVisu/description.pdf"/>
28          </dataref>
29          <mime type="application/pdf" application="Adobe Acrobat"/>
30        </node>
31        …
32      </graph>
33    </GraphXML>
```

The file contains what is called an “internal DTD” (lines 15–21), which extends the possible elements that can be included in a node definition. The definition states that a `<mime>` element can be added as the child of a node, that

this is an element that contains only attributes (i.e. no sub–elements), and that the attributes can be `type` and `application` (the first being compulsory, the second optional)*.

The syntax is a bit cryptic: this is a side–effect of the complexity of XML. However, the end–user does not necessarily have to include such internal DTD's into all his/her GraphXML files. Instead, using standard XML mechanisms, it is possible to define a *separate* DTD file containing the application specific extensions (and a reference to the `GraphXML.dtd`, of course). Using such extension, the example becomes simply:

```
1    <?xml version="1.0"?>
2    <!DOCTYPE GraphXML SYSTEM "file: WebVisualizer.dtd ">
3    <GraphXML>
4      <graph>
5        <node name="first">
6          <label>Project Home page</label>
7          <dataref>
8            <ref xlink:href="http://www.cwi.nl/InfoVisu/description.pdf"/>
9          </dataref>
10         <mime type="application/pdf" application="Adobe Acrobat"/>
11       </node>
12       …
13     </graph>
14   </GraphXML>
```

The only difference is in line 2, where a reference to `WebVisualizer.dtd`, replaces `GraphXML.dtd`. In other words, the intricacies of the XML DTD syntax remain invisible to most users. The authors of the application can extend the functionalities of GraphXML once and for all. Furthermore, because the extension is made through a standard XML mechanism, the basic GraphXML parser remains unchanged, only the application–dependent part has to be adapted for the new extensions. Section 7.10 describes the application specific DTD's in more details.

6 APPLICATION CONTROL

A single GraphXML file can be used in many different visualization systems, each with its own special features. It is very useful if the description of a graph could contain a statement such as "if displayed in this application, use these flags, if displayed in that application, use those flags", etc.

The standard XML processing instructions can be used for that purpose. An XML file can contain the following lines:

```
1    <?ApplicationName Parametersfortheapplication ?>
```

An XML processor does not interpret these lines directly, but tries to identify the application `ApplicationName`, and forward the parameters (i.e. all characters between the application name and the closing "?>" sequence).†

This XML feature can be used in GraphXML. As a convention, the parser of GraphXML should identify whether the application, which has invoked the parser, is `ApplicationName`; if yes, the parameters should be forwarded to the application, otherwise the processing instruction is ignored. For example, if the GraphXML file is parsed by the application called Royere, which is a graph visualization system that was developed at CWI, the following line:

```
1    <?Royere Timer=true ?>
```

will instruct the program to turn its debug timer on during the display of that graph.

Of course, the processing instructions can be included in any separate DTD file, just as user extensions can be, which provides flexible control of applications.

---

* Actually, extending the GraphXML specification with elements admitting some predefined attributes only is so frequent, that a separate tag, called <properties> is pre–defined in GraphXML, so that the internal DTD should define the attribute list only (like in lines 17–20, although with a slightly different syntax. See Section 7 for more details).

† The only restriction is that the application name may not begin with the string 'xml'. This is reserved by W3C.

7 DETAILED SPECIFICATION OF GRAPHXML

This section contains a more detailed specification of the XML tags defined in GraphXML. Each element is described in detail. Familiarity with the basic concepts of XML is necessary to understand this section. The DTD extracts below are not the exact copies of the real DTD. Some attributes, whose values are fixed but required by, for example, XML or XLink, have been omitted to simplify the description. The "real" DTD appears in the Appendix.

7.1 Common elements

To simplify the description and the DTD, some "common elements" have been factored out into a separate DTD entity. These are the application dependent data (see Section 2.2), styles (see Section 4), and extensible properties (see Section 7.10.1 below). These elements can be used for graphs, nodes, and edges. They must precede the "real" content of the elements, such as the list of nodes and edges. Details on these elements will be given in Section 7.8 below, but to simplify the forthcoming specification, only the entity itself is described here:

```
<!ENTITY % common-elements "label|data|dataref|properties">
```

The entity specification allows, gramatically, to have more than one common element of the same type as a child. For example, it is possible to have several labels added to a node. Although the XML syntax allows this, the semantics of GraphXML does not: the first element overrides all the others, and the remaining elements (e.g., the second label) is ignored.

7.2 The root tag: GraphXML

A GraphXML file can consist of several graph specifications, as well as several edit or edit bundle specifications.

```
<!ELEMENT GraphXML ((%common-elements;|style)*,graph*,(edit|edit-bundle)*)>
```

7.3 The graph tag

Beyond the common elements, the graph element can also use the <icon> and <size> elements as part of its introductory tags. The first is simply a reference to an image file and can be used by the application to, e.g., control a window icon; the second has been described in Section 3.3. Furthermore, a <style> element can also be added, as described in Section 4. Otherwise, a graph is simply a collection of nodes and edges.

The attributes are straightforward. Note that the value of the id attribute must be unique within a file; it is used as a reference target for hierarchical graph specifications.

```
<!ELEMENT graph ((%common-elements;|style|icon|size)*,(node|edge)*)>
<!ATTLIST graph
    isDirected      (true|false) "true"
    isPlanar        (true|false) "false"
    isAcyclic       (true|false) "false"
    isForest        (true|false) "false"
    preferredLayout CDATA        #IMPLIED
    vendor          CDATA        #IMPLIED
    version         CDATA        #IMPLIED
    id              ID           #IMPLIED
>
```

```
<!ELEMENT icon EMPTY>
<!ATTLIST icon
    xlink:href CDATA #REQUIRED
>
```

7.4 The edit and the edit bundle elements

The semantics of these elements has been described in Section 2.4. The target of the `xlink:href` is a URL containing the `id` of the graph. The `xlink:role` can be a useful hint to the application, but it does not have any predefined semantics.

```
<!ELEMENT edit ((%common-elements;)*,(node|edge)*)>
<!ATTLIST edit
    action          (replace|remove)     #REQUIRED
    xlink:role      CDATA                #IMPLIED
    xlink:href      CDATA                #IMPLIED
>
```

```
<!ELEMENT edit-bundle (edit*)>
```

7.5 The node element

The name of a node must be unique within the graph statement that contains it (although not necessarily within a full GraphXML file). The `xlink:href` is used when the node is a metanode (i.e. the value of `isMetanode` is `true`) in which case it should refer to the `id` of a graph. As with the edit element, the `xlink:role` can be a useful hint to the application, but it does not have any predefined semantics. For details on hierarchical graphs, see Section 2.3.

```
<!ELEMENT node (%common-elements;|style|subgraph-style|position|size|transform)*>
<!ATTLIST node
    name            CDATA                #REQUIRED
    isMetanode      (true|false)         "false"
    xlink:role      CDATA                #IMPLIED
    xlink:href      CDATA                #IMPLIED
    class           CDATA                #IMPLIED
>
```

The `<style>` element can be added, as well as a `class` attribute, which is used to categorize nodes in terms of common visual properties for visual property control (see Section 4). If the node is a metanode, the `<subgraph-style>` element can be used to control the visual appearance of the graph referred to by the `xlink:href` attribute.

A node can also specify its position and size (see Section 3.1). Furthermore, if the node is a metanode, the `<transform>` tag can be used to define a transformation for the referred graph to the current node. See Section 3.4 for further details. Note that the value of the `<size>` element has no semantic meaning in such a case (the size of the node is determined by the transformed bounding box of the referred graph).

```
<!ELEMENT transform EMPTY>
<!ATTLIST transform
    matrix  CDATA   "1.0 0.0 0.0 0.0  0.0 1.0 0.0 0.0  0.0 0.0 1.0 0.0"
>
```

7.6 The edge element

An edge tag is relatively simple: it contains reference attributes to the source and target nodes. The name is optional.

```
<!ELEMENT edge (%common-elements;|style|path)*>
<!ATTLIST edge
    name    CDATA       #IMPLIED
    source  CDATA       #REQUIRED
    target  CDATA       #REQUIRED
    class   CDATA       #IMPLIED
>
```

The geometry of the edge can be specified as a sequence of geometric positions, describing either a polyline, a spline, or a (circular) arc. In the case of an arc only the first three coordinate values are considered.

```
<!ELEMENT path (position)*>
<!ATTLIST path
    type    (polyline|spline|arc)   "polyline"
>
```

The <style> element can also be added, as well as a class attribute, which is used to categorize edges in terms of common visual properties, and is used for visual property control (see Section 4).

7.7 Geometry specifications

Two geometry specifications are used by nodes and graphs: the <size> and <position> tags. Both are useable for 3D visualization, but defaults are specified in a way that a purely 2D environment can function without any further problems.

```
<!ELEMENT size EMPTY>
<!ATTLIST size
    width   CDATA     #REQUIRED
    height  CDATA     #REQUIRED
    depth   CDATA     "0.0"
>
```

```
<!ELEMENT position  EMPTY>
<!ATTLIST position
    x         CDATA   "0.0"
    y         CDATA   "0.0"
    z         CDATA   "0.0"
>
```

7.8 Specification of application data

There are three categories of application dependent data (see Section 2.2): labels, data, and data references. The first two are simple tags, whereas the third is a collection of external references.

```
<!ELEMENT label (#PCDATA)>
<!ATTLIST label
    class       CDATA        #IMPLIED
>
```

```
<!ELEMENT data (#PCDATA)>
<!ATTLIST data
    class       CDATA        #IMPLIED
>
```

```
<!ELEMENT dataref (ref*)>
```

Note that in the GraphXML DTD, the <dataref> tag is defined as an “extended” XLink element; in other words the tag, and its immediate sub tags, can be manipulated as one element. This also explains why editing a data reference tag through the <edit> tags can be done as a unit only (see Section 2.4).

```
<!ELEMENT ref EMPTY>
<!ATTLIST ref
    xlink:role      CDATA               #IMPLIED
    xlink:href      CDATA               #REQUIRED
>
```

7.9 Visual attribute control

Visual attributes are controlled via the `<style>` element, which can be added to graphs, nodes, and edges. The `<subgraph-style>` has a similar syntax, although it can be used as a child of a metanode only.

```
<!ELEMENT style          (line|fill|implementation)*>
<!ELEMENT subgraph-style (line|fill|implementation)*>
```

7.9.1 Line attributes

The `<line>` element controls the appearance of the node glyph borders (if the `tag` attribute is set to `node`) or the edge (if the `tag` attribute is set to `edge`). The `tag` attribute is ignored if the `<style>` tag, containing the `<line>` tag, is a direct child of either a node or an edge. The `class` attribute is a selector for the category of nodes or edges.

```
<!ELEMENT line EMPTY>
<!ATTLIST line
    tag          (edge|node)     "edge"
    class        CDATA           #IMPLIED
    color_start  CDATA           #IMPLIED
    color_end    CDATA           #IMPLIED
    colour_start CDATA           #IMPLIED
    colour_end   CDATA           #IMPLIED
    color        CDATA           #IMPLIED
    colour       CDATA           #IMPLIED
    linestyle    CDATA           #IMPLIED
    linewidth    CDATA           #IMPLIED
>
```

Colour setting can be done either through the `color` or the `colour` attributes. If both appear in the tag, the color tag is ignored. The value of the attribute can be:

- one of `black`, `blue`, `cyan`, `darkGray`, `gray`, `green`, `lightGray`, `magenta`, `orange`, `pink`, `red`, `white`, `yellow`, to denote a specific colour, or
- a 24 bit hexadecimal number, using the format `#rrggbb`, to set explicit red, green, an blue values, or
- a 32 bit hexadecimal numb er, using the format `#rrggbbaa`, to set explicit red, green, blue, and alpha values.

Alternatively, the colour can be set independently to the start and the end of the edge through the `colour_start`, `colour_end` (or, alternatively, through the `color_start` and `color_end`) attributes. The values for these attributes is identical to the simple colour setting. Semantically, these attributes set the colours on the two end of the edges and the application is supposed to provide a smooth transition of the colours along the edge. These settings have a higher priority than the simple colour setting: if they are all set, the `colour` attribute value is ignored.

The linestyle attribute can be:

- one of `none`, `solid`, `dashed`, `dash-dotted`, `dotted` to denote a specific linestyle, or
- a 16 bit hexadecimal number, using the format `#bbbb`, to define a bit pattern to be used when drawing the line.

7.9.2 Fill attributes

The `<fill>` element controls the appearance of the node glyph's interior (if the `tag` attribute is set to `node`) or the interior of the edge (if the `tag` attribute is set to `edge`). The `tag` attribute is ignored if the `<style>` tag, containing the `<fill>` tag, is a direct child of either a node or an edge. Note that if edges are rendered with lines, the fill element may have no effect on the edge at all.

```
<!ELEMENT fill EMPTY>
<!ATTLIST fill
    tag             (edge|node)             "node"
    class           CDATA                   #IMPLIED
    color           CDATA                   #IMPLIED
    colour          CDATA                   #IMPLIED
    fillstyle       (solid|none|background) #IMPLIED
    xlink:href      CDATA                   #IMPLIED
    imagefill       (resize|duplicate|none) #IMPLIED
>
```

The interpretation of the `color` and `colour` tags is similar to line control. The `fillstyle` attribute determines whether the fill should be done with the specified (fill) colour, whether no fill should be performed, or whether the background colour should be used.

The `xlink:href` refers to an image, which can be used as simply an image to be displayed, or as a fill pattern (depending on the capabilities of the underlying visualization system). If specified, filling with a pattern or image takes precedence over the `fillstyle` and `colour` attributes.

If the node also specifies its size (through the `<size>` tag), the `imagefill` attribute controls whether the content of the image must be resized or duplicated to fill the full size, or whether the `<size>` tag should be ignored and the image displayed with its original size. The `duplicate` value is usually used if the image file is a pattern.

7.9.3 Direct implementation control

In some cases, the user might want to have complete control over the implementation of the node and/or the edge visualization through a script, program applet, Java class, etc. Details of how this can be done is highly environment and visualization system dependent. The `<implementation>` tag simply provides the name of a script or class object which the application can then interpret and invoke. If the `<implementation>` tag is valid for a node or edge, this overrides all other tags, although the visualization system should convey all other visual properties to the corresponding class or script.

```
<!ELEMENT implementation EMPTY>
<!ATTLIST implementation
    tag         (edge|node)     #REQUIRED
    class       CDATA           #IMPLIED
    scriptname  CDATA           #IMPLIED
>
```

7.10 User extensions

The user extension mechanism in GraphXML is based on entities, which are part of the DTD specifications but whose (initial) value is empty. The extension is done by giving appropriate values to those entities, and defining, if necessary, other tags in the internal DTD of the GraphXML file.This extension mechanism relies on two important features of XML DTD's:

1. If, in parsing a DTD, the same definition (for an element, entity, attributes, etc.) is encountered twice, the second occurrence is silently ignored.
2. The internal DTD portion is read and parsed *before* the external DTD.

Two mechanisms for extension are built into the GraphXML DTD: adding properties to tags and adding new tag specification to existing tags.

7.10.1 Adding properties

The common elements (see Section 7.1) include the <properties> tag, defined as:

```
<!ENTITY % admissibleProperties "">
<!ELEMENT properties EMPTY>
<!ATTLIST properties
    %admissibleProperties;
>
```

This means that this <property> tag can be used as a direct descendent to all nodes and edges. Because the value of admissibleProperties is empty in the GraphXML DTD (i.e. the tag does not have any content or valid properties), this tag cannot be used until further definitions are available.

However, if an internal DTD is used, such as in the following example:

```
1    <?xml version="1.0"?>
2    <!DOCTYPE GraphXML SYSTEM "file:GraphXML.dtd" [
3    <!ENTITY % admissibleProperties "
4        attributeA (value1|value2) #IMPLIED
5        attributeB CDATA           #IMPLIED
6    ">
7    ]>
8    <GraphXML>
9      …
10   </GraphXML>
```

then the entity admissibleProperties defines the attributes attributeA and attributeB, which can be used within the GraphXML file itself. This is an easy way of adding properties to all tags in a GraphXML file.

7.10.2 Adding new tags

The GraphXML DTD also defines hooks for extending admissible tags for the root level, for nodes, edges, graphs, and edit tags. These hooks are also based on the use of entities and enable the addition of properties.

Using the <node> tag as an example, the “real” specification of this tag in the GraphXML DTD is:

```
<!ENTITY % nodeExtensions"">
<!ELEMENT node (%common-elements;%nodeExtensions;|position|size|transform)*>
```

in contrast to the specification given in Section 7.5. Because the initial value for nodeExtension is empty, this entity initially has no effect on <node>. However, if the entity receives an initial value in an internal DTD, such as in the example in Section 5, then new tags can be used and the GraphXML file remains valid. Similar mechanisms are defined using the entities rootExtensions, edgeExtensions, graphExtensions, and editExtensions.

Note that the XLink namespace is added to the DTD on the GraphXML level. This means that all new tags that are defined by the user can use all the xlink:xxx attributes as well.

8 TRANSFORMING GRAPHXML FILES

The primary goal of GraphXML is to describe graphs which are to be visualized by specialized applications. However, one can use the XSLT mechanism[3,13] to dynamically transform a GraphXML file into, for example, an HTML file, resulting in some pretty printing format of a graph specification. To include an XSLT reference in the graph file, the following processing instruction should be used:

```
1    <?xml-stylesheet href="YourCSSFile.xsl" title="You styles" type="text/xsl"?>
```

With the evolution of Web standards, this transformation mechanism can also be used for more powerful ends. For example, if XML vocabularies to describe graphics become available, it will be possible to interpret the geometric attributes directly and give a visual representation of the graphs through a web browser.

9 HOW TO HIDE INTERNAL DTD'S

The user extension mechanism of GraphXML is based on the use of internal DTD's, which do not have a very friendly syntax. However, XML entities can be used to hide them. For example, this is how the `WebVisualizer.dtd` file, referred to in Section 5, might appear:

```
1    <!ENTITY % GraphXMLDTD SYSTEM "GraphXML.dtd">
2    <!ENTITY % nodeExtensions ",mime*">
3    <!ELEMENT mime EMPTY>
4    <!ATTLIST mime
5        type        CDATA #REQUIRED
6        application CDATA #IMPLIED
7    >
8    %GraphXMLDTD;
```

The trick is to refer to the "real" DTD as an external entity, which is included in the file only at line 8 (when the specification for the `<mime>` tag is already parsed). Similarly, company–wide style files can be created. By storing the `<style>` nodes and processing instructions in a separate file called, for example, "`CWIStyle.xml`", a separate, "pseudo" DTD file can be defined as:

```
1    <!ENTITY % GraphXMLDTD SYSTEM "GraphXML.dtd">
2    %GraphXMLDTD;
3    <!ENTITY CWIStyle SYSTEM "CWIStyle.xml">
```

By calling this file, for example, `GraphXMLWithStyle.dtd`, the GraphXML file would appear as follows:

```
1    <?xml version="1.0"?>
2    <!DOCTYPE GraphXML SYSTEM "file:GraphXMLWithStyle.dtd">
3    <GraphXML>
4        <graph>
5            &CWIStyle;
6            <node name="first"/>
7            <node name="second" class="special"/>
8            …
9            <node name="nth" class="special"/>
10           …
11           <edge source="first" target="second"/>
12           …
13       </graph>
14   </GraphXML>
```

Where the entity reference in line 5 includes the `<style>` content or the processing instructions. Using this mechanism, the file `CWIStyle.xml` plays the role of a general property file for a specific application.

10 USING NON-VALIDATING PARSERS

All examples listed so far include a reference to the GraphXML DTD. If the parser used by the application is a "validating" parser, this is necessary. However, there might be applications that are not so strict and restrict themselves to a weaker form of checking, called "well–formedness" in the XML jargon (the term "non-validating parser" is also used in this context). What this means is that the parser would check for the syntax rules (are all opening elements closed, etc.), but not for the stricter rules such as not allowing a `<subgraph-style>` element to appear as a child of `<edge>`. Using a non-validating parser has a number of advantages. For example, the extension mechanisms, described in Section 7.10, become much simpler: any new elements and attributes can be added to the graph. If the application "understands" those extensions, they can be used without any trouble. Another problem might occur if the DTD file itself is not accessible to the user (because, for example, the Internet location is not accessible). On the other hand, using a non–validating parser has obvious disadvantages, too. No checks are done on the elements and attributes, which could lead to the misinterpretation of the data due to spelling mistakes.

11 PUBLIC AVAILABILITY

This document along with the GraphXML.dtd file, some examples, and the public domain parser for GraphXML can be retrieved at the following URL: http://www.cwi.nl/InfoVisu/GraphXML.

The parser itself is a collection of Java 1.2 classes. The current implementation version relies on Version 3 of the XML4J parser, which is freely downloadable from the IBM: http://www.alphaworks.ibm.com/tech/xml4j, the previous (Version 2) version of the XML4J parser, which users may still have, or Version 1.0 of the JAXP parser, which is freely downloadable from SUN: http://www.sun.com/xml. Version 3 of XML4J is identical to the 1.0.3 version of the Apache XML parser, codenamed Xerces, and is also available from http://xml.apache.org/xerces-j/index.html. All these parsers are 100% Java parsers, and are easy to install on any platform where Java 1.2 is available. Depending on a system property `XMLGraph_Validate`, the parser can be used either in validating or non–validating mode: if the value of the property is set to `No`, `no`, `False`, or `false`, the parser is non–validating, and is validating otherwise. The latter is the default.

The GraphXML parser checks syntax as well as some general semantics. Different applications can be bound to the parser by providing an implementation of the `GraphSemantics.java` interface. A simple example for an implementation is available in the `debug` package of the distribution.

REFERENCES

1. "Extensible Markup Language (XML) 1.0", World Wide Web Consortium, (eds. T. Bray, J. Paoli, C.M. Sperberg–McQueen), Recommendation February 1998, http://www.w3.org/TR/REC-xml.

2. N. Bradley, *The XML Companion*,. Harlow: Addison Wesley Longman, 1998.

3. W. J. Pardi, *XML in Action*,. Redmond: Microsoft Press, 1999.

4. M. Himsolt, *GML — Graph Modelling Language*, University of Passau, http://infosun.fmi.uni-passau.de/Graphlet/GML/, 1997.

5. S. C. North, *DOT abstract graph description format*, http://www.research.att.com/~north/cgi-bin/webdot.cgi/dot.txt .

6. "Document Object Model (DOM) Level 1 Specification", World Wide Web Consortium, (eds. V. Apparao *et al*.), Recommendation October 1998, http://www.w3.org/TR/REC-DOM-Level-1/.

7. "XML Linking Language (XLink)", World Wide Web Consortium, (eds. S. DeRose, D. Orchard, B. Trafford), Working Draft July 1999, http://www.w3.org/TR/WD-xlink.

8. P. Eades and Q.-W. Feng, "Multilevel Visualization of Clustered Graphs", in Proceedings of Symposium on Graph Drawing GD '96, Berlin, 1997.

9. D. Schaffer, Z. Zuo, S. Greenberg, L. Bartram, J. Dill, S. Dubs, and M. Roseman, "Navigating Hierarchically Clustered Networks through Fisheye and Full–zoom Methods", *ACM Transactions on Computer–Human Interaction*, vol. 3, pp. 162–188, 1996.

10. I. Herman, S. Marshall, and G. Melançon, "Graph Visualisation and Navigation in Information Visualisation: a Survey", To be published in: IEEE Transactions on Visualization and Computer Graphics, vol. 6, 2000. An earlier version is also available in: Proceedings of Eurographics '99 State–of–the–Art Reports, Milano, Italy, 1999, http://www.cwi.nl/InfoVisu/Survey/StarGraphVisuInInfoVis.pdf.

11. "XML Pointer language (XPointer)", World Wide Web Consortium, (eds. S. DeRose and R. Daniel Jr.), Working Draft July 1999, http://www.w3.org/TR/WD-xptr.

12. J. D. Foley, A. v. Dam, and S. K. Feiner, *Computer Graphics: Principles and practice*,. Reading: Addison–Wesley, 1990.

13. “XSL Transformations”, World Wide Web Consortium, (ed. J. Clark), Recommendation November, 1999, http://www.w3.org/TR/xslt.

Appendix 1 THE GRAPHXML DTD

```
<!ENTITY % common-elements "label|data|dataref|properties">

<!ENTITY % rootExtensions "">
<!ENTITY % nodeExtensions "">
<!ENTITY % edgeExtensions "">
<!ENTITY % graphExtensions "">
<!ENTITY % editExtensions "">

<!ENTITY % admissibleProperties "">

<!-- ================================================================== -->
<!-- ================================================================== -->

<!ELEMENT GraphXML
    ((%common-elements;%rootExtensions;|style)*,graph*,(edit|edit-bundle)*)>
<!ATTLIST GraphXML
    xmlns:xlink  CDATA   #FIXED   "http://www.w3.org/XML/XLink/0.9"
>

<!-- ================================================================== -->
<!-- ================================================================== -->

<!ELEMENT graph
           ((%common-elements;%graphExtensions;|style|icon|size)*,(node|edge)*)>
<!ATTLIST graph
    isDirected      (true|false) "true"
    isPlanar        (true|false) "false"
    isAcyclic       (true|false) "false"
    isForest        (true|false) "false"
    preferredLayout CDATA        #IMPLIED
    vendor          CDATA        #IMPLIED
    version         CDATA        #IMPLIED
    id              ID           #IMPLIED
    class           CDATA        #IMPLIED
>

<!ELEMENT icon EMPTY>
<!ATTLIST icon
    xlink:type      CDATA               #FIXED   "simple"
    xlink:role      CDATA               #FIXED   "Icon image for the full graph"
    xlink:title     CDATA               #IMPLIED
    xlink:show      (new|parsed|replace) #FIXED   "parsed"
    xlink:actuate   (user|auto)         #FIXED   "auto"
    xlink:href      CDATA               #REQUIRED
>

<!ELEMENT edit ((%common-elements;%editExtensions;)*,(node|edge)*)>
<!ATTLIST edit
    action          (replace|remove)    #REQUIRED
    xlink:type      CDATA               #FIXED   "simple"
    xlink:role      CDATA               #FIXED   "Reference to graph"
    xlink:title     CDATA               #IMPLIED
    xlink:show      (new|parsed|replace) #FIXED   "new"
    xlink:actuate   (user|auto)         #FIXED   "user"
    xlink:href      CDATA               #IMPLIED
    class           CDATA               #IMPLIED
>
```

```
<!ELEMENT edit-bundle (edit*)>
<!ATTLIST edit-bundle
    class       CDATA        #IMPLIED
>

<!ELEMENT node
    (%common-elements;%nodeExtensions;|style|subgraph-style|position|size|transform)*>
<!ATTLIST node
    name            CDATA               #REQUIRED
    isMetanode      (true|false)        "false"
    xlink:type      CDATA               #FIXED   "simple"
    xlink:role      CDATA               #FIXED   "Reference to graph"
    xlink:title     CDATA               #IMPLIED
    xlink:show      (new|parsed|replace) #FIXED   "new"
    xlink:actuate   (user|auto)         #FIXED   "user"
    xlink:href      CDATA               #IMPLIED
    class           CDATA               #IMPLIED
>

<!ELEMENT edge (%common-elements;%edgeExtensions;|style|path)*>
<!ATTLIST edge
    name    CDATA       #IMPLIED
    source  CDATA       #REQUIRED
    target  CDATA       #REQUIRED
    class   CDATA       #IMPLIED
>

<!ELEMENT properties EMPTY>
<!ATTLIST properties
    class   CDATA   #IMPLIED
    %admissibleProperties;
>

<!ELEMENT label (#PCDATA)>
<!ATTLIST label
    class       CDATA        #IMPLIED
>

<!ELEMENT data (#PCDATA)>
<!ATTLIST data
    class       CDATA        #IMPLIED
>

<!ELEMENT dataref (ref*)>
<!ATTLIST dataref
    xlink:type      CDATA       #FIXED   "extended"
    xlink:role      CDATA       #FIXED   "Reference to external application data"
    xlink:title     CDATA       #IMPLIED
    xlink:show      (new|parsed|replace) #FIXED   "new"
    xlink:actuate   (user|auto)         #FIXED   "user"
    class           CDATA               #IMPLIED
>

<!ELEMENT ref EMPTY>
<!ATTLIST ref
    xlink:type      CDATA               #FIXED   "locator"
    xlink:role      CDATA               #IMPLIED
    xlink:title     CDATA               #IMPLIED
    xlink:show      (new|parsed|replace) #FIXED   "new"
    xlink:actuate   (user|auto)         #FIXED   "user"
    xlink:href      CDATA               #REQUIRED
    class           CDATA               #IMPLIED
>
```

```
<!ELEMENT position  EMPTY>
<!ATTLIST position
    x         CDATA   "0.0"
    y         CDATA   "0.0"
    z         CDATA   "0.0"
    class     CDATA   #IMPLIED
>
<!ELEMENT size EMPTY>
<!ATTLIST size
    width   CDATA     #REQUIRED
    height  CDATA     #REQUIRED
    depth   CDATA     "0.0"
    class   CDATA     #IMPLIED
>

<!ELEMENT path (position)*>
<!ATTLIST path
    type    (polyline|spline|arc)   "polyline"
    class   CDATA                   #IMPLIED
>

<!ELEMENT transform EMPTY>
<!ATTLIST transform
    matrix  CDATA   "1.0 0.0 0.0 0.0  0.0 1.0 0.0 0.0  0.0 0.0 1.0 0.0"
    class   CDATA   #IMPLIED
>

<!ELEMENT style          (line|fill|implementation)*>
<!ELEMENT subgraph-style (line|fill|implementation)*>

<!ELEMENT line EMPTY>
<!ATTLIST line
    tag          (edge|node)     "edge"
    class        CDATA           #IMPLIED
    color_start  CDATA           #IMPLIED
    color_end    CDATA           #IMPLIED
    colour_start CDATA           #IMPLIED
    colour_end   CDATA           #IMPLIED
    color        CDATA           #IMPLIED
    colour       CDATA           #IMPLIED
    linestyle    CDATA           #IMPLIED
    linewidth    CDATA           #IMPLIED
>

<!ELEMENT fill EMPTY>
<!ATTLIST fill
    tag             (edge|node)             "node"
    class           CDATA                   #IMPLIED
    color           CDATA                   #IMPLIED
    colour          CDATA                   #IMPLIED
    fillstyle       (solid|none|background) #IMPLIED
    xlink:type      CDATA                   #FIXED   "simple"
    xlink:role      CDATA                   #FIXED   "Fill image or pattern"
    xlink:title     CDATA                   #IMPLIED
    xlink:show      (new|parsed|replace)    #FIXED   "parsed"
    xlink:actuate   (user|auto)             #FIXED   "user"
    xlink:href      CDATA                   #IMPLIED
    imagefill       (resize|duplicate|none) #IMPLIED
>
```

```
<!ELEMENT implementation EMPTY>
<!ATTLIST implementation
    tag         (edge|node)     #REQUIRED
    class       CDATA           #IMPLIED
    scriptname  CDATA           #IMPLIED
>
```