GRAPH FILE FORMATS

There are many different file formats for graphs. The capabilities of these file formats range from simple adjacency lists or coordinates to complex formats that can store arbitrary data. This has lead to an almost "babylonic" situation where we have a large number of different, mostly incompatible formats. Exchanging graphs between different programs is painful, and sometimes impossible. The obvious answer to this problem is the introduction of a common file format.

Why do programs still use their own formats?

One reason is that exchange formats often do not support all product and platform specific features. This is inevitable, but should not exclude the exchange of platform independent parts, probably with a less-efficient, portable replacement for product specific features. Another concern is efficiency. One should not expect a universal format to be more efficient than one that is designed for a specific purpose, but there is no reason that a common file format should be so inefficient that it cannot be used. In the case of graphs, many file formats for graphs are not designed for efficiency, but for ease of use, so the overhead should be small. Furthermore, there is no reason that prevents the use of both an optimized native format, and a second interchange format.

Which features are necessary for a common file format?

1. The format must be **platform independent**, and **easy to implement**.
2. It must have the capability to represent **arbitrary data structures**, since advanced programs have the need to attach their specific data to nodes and edges.
3. It should be **flexible** enough that a specific order of declarations is not needed, and that any non-essential data may be omitted.

With this impending problem, the Graph Drawing community through the Symposia on Graph Drawing (GD ’XX conferences) agreed to introduce a common file format. This report discusses several graph file formats have been proposed.

GRAPH MODELLING LANGUAGE (GML)

GML, the **G**raph **M**odelling **L**anguage [1, 2] is a file format for graphs whose key features are portability, simple syntax, extensibility and flexibility. GML is designed to represent arbitrary data structures. A GML file consists of hierarchical key-value lists. GML was bound to a specific system, namely Graphlet [3, 4]. However, it has been overtaken and adapted by several other systems for drawing graphs.

GML possesses the following attributes in its attempt to satisfy the common file format requirements:

- **ASCII Representation for Simplicity and Portability**
  A GML file is a 7-bit ASCII file. This makes it simple to write files through standard routines. Parsers are easy to implement, either by hand or with standard tools like `lex` and `yacc`. Files are text files and can be exchanged amongst platforms without special converters.

- **Simple Structure**
  A GML file consists of hierarchically organized key-value pairs. A key is a sequence of alphanumeric characters, such as graph or id. A value can be an integer, a floating-point number, a string or a list of key-value pairs enclosed in square brackets. GML can be used to represent most Common data types and Data Structures such as: Integers, Floating points Boolean, Pointers, Records, Lists, Sets and Arrays.

- **Extensibility & Flexibility**
  GML can represent arbitrary data with the option to attach additional information to every object. For example, the graph in Figure 1 below adds an `IsPlanar` attribute to the graph. This can result in situations where an application adds data that cannot be understood by another application. Therefore, applications are free to ignore any data that they do not understand. They should, however, save these data and re-write them.

Representation of Graphs

Graphs are represented by the keys `graph`, `node` and `edge`. The topological structure is modelled with the node's `id` and the edge's `source` and `target` attributes: the `id` attributes assign numbers to nodes, which are referenced by `source` and `target`.

```
graph [
  node [
    id 7
    label "5"
    edgeAnchor "corners"
    labelAnchor "n"
    graphics [
      center [ x 82.0000 y 42.0000 ]
      w 16.0000
      h 16.0000
      type "rectangle"
      fill "#000000"
    ]
  ]
  node [
    id 15
    label "13"
    edgeAnchor "corners"
    labelAnchor "c"
    graphics [
      center [ x 73.0000 y 160.000 ]
      w 16.0000
      h 16.0000
      type "rectangle"
      fill "#FF0000"
    ]
  ]
  edge [
    label "24"
    labelAnchor "first"
    source 7
    target 15
    graphics [
      type "line"
      arrow "last"
      Line [
        point [ x 82.0000 y 42.0000 ]
        point [ x 10.0000 y 10.0000 ]
        point [ x 100.000 y 100.000 ]
        point [ x 80.0000 y 30.0000 ]
        point [ x 120.000 y 230.000 ]
        point [ x 73.0000 y 160.000 ]
      ]
    ]
  ]
]
```

Figure 1: This graph is a edited text file which was generated by the Graphlet system.

GML Syntax

GML	::=	List
List	::=	(whitespace* Key whitespace^{+} Value)*
Value	::=	Integer | Real | String | **[** List **]**
Key	::=	[**a-z A-Z**] [**a-z A-Z 0-9**]*
Integer	::=	sign digit^{+}
Real	::=	sign digit* **.** digit* mantissa
String	::=	**"** instring **"**
sign	::=	*empty* | **+** | **-**
digit	::=	[**0-9**]
Mantissa	::=	*empty* | **E** sign digit
instring	::=	*ASCII* - {&,"} | & character^{+} ;
whitespace	::=	space | tabulator | newline

Figure 2: The GML Grammar in BNF Format.

A GML file defines a tree. Each node in the tree is labelled by a key. Leaves have integer, floating point or string values. The notion

$k_1.k_2. ... .k_n$

is used to specify a path in the tree where the nodes are labelled by keys k_1, k_2, ... k_n.

$x.k_1.k_2. ... .k_n$

is used to specify a path which starts at a specific node *x* in the tree.

In the above grammar, all lines starting with a "#" character are ignored by the parser. This is a standard behavior for most UNIX software and allows the embedding of foreign data in a file as well as within the GML structure. However, it is convenient to add large external data through this mechanism, as any lines starting with # will not be read by another application. The above grammar is kept as simple as possible. Keys and values are separated by white space. With that, it is straightforward to generate a GML file from a given structure, and a parser can easily be implemented on various platforms.

With GML, a graph is defined by the keys *`graph`*, *`node`* and *`edge`*, where *`node`* and *`edge`* are sons of *`graph`* in no particular order. Each non-isolated node must have a unique *.graph.node.id* attribute. Furthermore, the end nodes of the edges are given by the *.graph.edge.source* and *`.graph.edge.target`* attributes. Their values are the *graph.node.id* values of end nodes.

Directed and undirected graphs are stored in the same format. The distinction is done with the *`.graph.directed`* attribute of a graph. If the graph is undirected that attribute is omitted. In an undirected graph, *`.graph.edge.source`* and *`.graph.edge.target`* may be assigned arbitrarily. GML does not define separate representations for directed and undirected graphs since it would have made the parser more complex, especially in applications that read both directed and undirected graphs and additionally if graphics are being used source and target have a meaning even for undirected graphs for example, if an edge is represented by a polyline, then the sequence of points implies a direction on the edge.

GML does usually not require that attributes appear in a specific order in the file. The order of objects is not considered significant *as long as their keys are different*. That is, if there are several attributes with the same key (id, label) in a list, then the parser integrated into must preserve their order.

GML is designed so that any application can add its own features to graphs, nodes and edges. However, not all applications understand all attributes. GML deals with foreign data in two ways:

1. Simply ignore it. However, this means the data gets lost when the file is written, for example, a program that does graph transformations would throw away any graphics data.
2. An even greater complication is to save everything to a generic structure and write it back when a new file is written. This may guarantee no data is lost but can result in inconsistencies if the application alters the graph since both changes in the structure and in the values of attributes can make other attributes invalid.

GML specifies a way in which attributes are safe with and without changes through the following rule:

> Any keyword that starts with a capital letter should be considered invalid as soon as any changes have occurred. We call such a key *unsafe*.

Restrictions

1. The values of the `.graph.node.id` elements must be unique *within the graph*.
2. Each edge must have `.graph.edge.source` and `.graph.edge.target` attributes.
3. Not all nodes have a `.id` field since this field is considered not necessary for isolated nodes. Referencing the node can be problematic.
4. With these conventions, a simple parser for a Graph in GML works in four steps:

    1. Read the file and build the tree.
    2. Scan the tree for a node *g* labeled `graph`.
    3. Find and create all nodes in `g.node`. Remember their `g.node.id` values.
    4. Find all edges in `g.edge`, and their `g.edge.source` and `g.edge.target` attributes. Find the end nodes and insert the edges.

    Step 1 should be integrated into the other steps to gain efficiency. It requires all attributes to be saved leading to overhead. However, extraction of data attached to nodes, edges and graphs, becomes easier more so to preserve unknown data.

5. Validation of the file is not possible using tools.

GML is a capable description language for graph drawing purposes and while it includes provision for extensions; the mechanisms for associating external data with a graph element is
provision for extensions; the mechanisms for associating external data with a graph element is
not well defined.

Since graphs can be described as a data object whose elements are nodes and edges that are data objects, XML is an ideal way to represent graphs. Structure of the World Wide Web is a typical example of a graph where the web pages are "nodes," and the hyperlinks are "edges." GML is not XML-based but its structure strongly resembles and follows XML format. All other file formats discussed are XML-based.

EXTENSIBLE GRAPH MARKUP AND MODELING LANGUAGE (XGMML)

Extensible Graph Markup and Modeling Language (XGMML) is an XML 1.0 application based on Graph Modeling Language (GML) to describe graphs. The best way to describe a web site structure is using a graph structure so XGMML documents are a good choice for containing the structural information of a web site.

Since XGMML documents are XML based, the documents must be:

1. Well formed: Two cases of XGMML well-formed document can be found.
    a. XGMML documents with additional proprietary elements from a vendor.
    b. XGMML documents that are contained on other XML documents.
2. Valid: An XGMML valid document can be validated against an XGMML DTD or XGMML Schema The namespace for XGMML is: *http://www.cs.rpi.edu/XGMML* and the suffix for the XGMML elements is *xgmml*:.

Structure of XGMML Documents

An XGMML document describes a graph structure. The root element is **graph** and it can contain **node, edge** and **att** elements. The **node** element describes a node of a graph and the **edge** element describes an edge of a graph. Additional information for graphs, nodes and edges can be attached using the **att** element. A **graph** element can be contained in an **att** element and this graph will be considered as subgraph of the main graph. The **graphics** element can be included in a **node** or **edge** element, and it describes the graphic representation either of a node or an edge. The following example is a graph with just one node.

Graph Element

```
<!ELEMENT graph (att*,(node | edge)*)>
<!ATTLIST graph
        %global-atts;
        %xml-atts;
        %xlink-atts;
        %graph-atts-safe;
        %graph-atts-gml-unsafe;
        %graph-atts-app-unsafe;>
```

The **graph** element is the root element of an XGMML valid document. This **graph** element contains the rest of the XGMML elements. The **graph** element may not be unique in the XGMML document. Other graphs can be included as subgraphs of the main graph. The only elements allowed in a **graph** element are: **node**, **edge** and **att**. The **graph** element can be an empty graph. For valid XGMML documents, **att**s may be placed first or last, and **node**s and **edge**s may be freely intermingled. Nodes must have different **id**s and **name**s attributes. Edges cannot reference to nodes that are not included in the XGMML definition.

The **graph** attributes can be safe or unsafe.

- **directed** - Boolean value. If value is 1 (true) graph is directed. Default value is 0 (false).
- **Vendor** - Unsafe GML key to show the application that created the XGMML file.
- **Scale** - Unsafe numeric value to scale the size of the displayed graph.
- **Rootnode** - Unsafe id number to identify the root node. Useful for tree drawing.
- **Layout** - Unsafe string that represents the layout that can be applied to display the graph. The layout name is the name of the algorithm used to assign positions to the nodes of the graph. For example: circular.
- **Graphic** - Unsafe boolean value. If value is 1 (true), the XGMML file includes a graphical representation of the graph. 0 (false) means that the XGMML file includes only the topological structure of the graph and the application program is free to display the graph using any layout.

Global Attributes

The following are attributes of all XGMML elements:

- **id** - Unique number to identify the elements of XGMML document
- **name** - String to identify the elements of XGMML document
- **label** - Text representation of the XGMML element
- **labelanchor** - Anchor position of the label related to the graphic representation of the XGMML element

Nodes and Edges can reference XGMML documents. For example, a node can represent a graph that can be shown when the user points to the node. This behavior is similar to hyperlinks in HTML documents. XGMML uses the XLink framework to create hyperlinks either in nodes or edges. All these attributes are taking directly from the XLink Working Draft.

Node Element

```
<!ELEMENT node (graphics?,att*)>
<!ATTLIST node
        %global-atts;
        %xlink-atts;
        %node-atts-gml-safe;
        %node-atts-app-safe;>
```

A **node** element must be included in a **graph** element. Each **node** element describes the properties of a node object. The only elements allowed inside the node are **graphics** and **att**. The node can be rendered as a graphic object, using **graphics** element and can also have additional meta information to be used for the application program using the **att** element. For example:

a) A graphical representation of a node can be a rectangle, a circle or a bitmap.
b) If a node represents a web page, useful metadata is title and date of creation.

The node attributes are:

- **edgeancho**r - GML key to position the edges related to the node
- **weight** - value (usually numerical) to show the node weight -Useful for weight graphs

Edge Element

```
<!ELEMENT edge (graphics?,att*)>
<!ATTLIST edge
        %global-atts;
        %xlink-atts;
        %edge-atts-gml-safe;
        %edge-atts-app-safe;>
```

An **edge** element must be included in a **graph** element. The **graphics** and **att** elements are the only elements allowed inside of the **edge** element. For each **edge** element, at least two **node** elements have to be included in the **graph** element. The application program must verify if the source node and target node are included in the XGMML document. The edge element as the node element can have a graphical representation and additional metadata information. For example;

a) a graphical representation of an edge can be a line or an arc.
b) If an edge represents a hyperlink, useful metadata is anchor string and type of hyperlink.

An optional attribute of an edge is its weight.

Att Element

```
<!ELEMENT att (#PCDATA | att | graph)*>
<!ATTLIST att
        %global-atts;
        %attribute-value;
        %attribute-type;>
```

An **att** element is used to hold meta information about the element that contains the **att** element. It can contain other **att** elements for example to represent structured metadata such as records.

The att attributes are:

- **name** - Global attribute that contains the name of the metadata information.
- **value** - The value of the metadata information.
- **type** - The object type of the metadata information. The default object type is string.

All of **att**, **graph** and PCDATA can be inside of **att** element. An **att** is an empty element for object types such as integers, reals and integers. When the object type is a list, other **att** element must be inside of the **att** element to hold the list information.

For example, the metadata of an object person A is name:John, ssn: 123456789 and e-mail:john@rpi.edu. To attach this metadata to a node of a graph using the **att** element, the following lines must be included in the **node** element:

```
<att type="list" name="person_description">
<att name="name" value="John"/>
<att name="ssn" value="123456789"/>
<att name="e-mail" value="john@rpi.edu"/>
</att>
```

Graphics Attributes

Line, **center** and **att** elements are the only elements that can be contained in a **graphics** element. **Line** element is defined between two **point** elements and it is used to represent edges. **center** element is a special **point** element to represent the central point of the graphical representation of a node. The **att** element permits to add information to the graphical representation. All these elements are inherited from GML.

The graphics attributes are divided in the following groups:

- Graphics type attribute: type
- Point attributes: **x**, **y** and **z** - The coordinates of a point (x, y, z). A point can be 2-or 3-D
- Dimension attributes: **w**, **h** and **d** - width, height and depth of the graphic object
- External attributes: image, bitmap
- Line attributes: width, arrow, capstyle, joinstyle, smooth, splinesteps
- Text attributes: justify, font
- Bitmap attributes: background, foreground
- Arc attributes: extent, arc, style
- Graphic object attributes: stipple, visible, fill, outline, anchor

The following is an example of a graph in both GML and XGMML format.

GML Format

```
graph [
   comment "This is a sample graph"
   directed 1
   id 42
   label "Hello, I am a graph"
   node [
      id 1
      label "Node 1"]
   node [
      id 2
      label "node 2"]
   node [
      id 3
      label "node 3"]
   edge [
      source 1
      target 2
      label "Edge from node 1 to node 2"]
   edge [
      source 2
      target 3
      label "Edge from node 2 to node 3"]
]
```

XGMML Format

```
<?xml version="1.0"?>
<!DOCTYPE graph PUBLIC "-//John Punin//DTD graph description//EN"
"http://www.cs.rpi.edu/~puninj/XGMML/xgmml.dtd">
<graph directed="1" id="42" label="Hello, I am a graph">
            <node id="1" label="Node 1">
            </node>
            <node id="2" label="node 2">
            </node>
            <node id="3" label="node 3">
            </node>
            <edge source="1" target="2" label="Edge from node 1 to node 2">
            </edge>
            <edge source="2" target="3" label="Edge from node 2 to node 3">
            </edge>
</graph>
```

Restrictions
There is a huge degree of redundancy in the grammar definition
Addition of any extra user defined attributes can only be included at a different conceptual level by using the **attr** element
A very shallow approach by putting so many attributes
where the grammar depth is great the storage overhead is significant.

PUT IN DISADVANTAGES OF SO MANY ATTRIBUTES

GRXL

GRXL draft version 0.1 is an XML DTD and worked example to show how a Grrr program might be stored in a form that allows conversion to other graph transformation systems. The approach adopted here explicitly includes the important attributes in the ATTLIST at the highest reasonable level. This approach allows for easier understanding of the resultant XML, and helps hand coding and editing.

A GRXL document is made up of one or more *attr*, *nodetype*, *edgetype*, *hostgraph* and *transformation* elements. All elements have a required *id* attribute.

nodetype: defines the type of node being created. A *nodetype* element is a collection of only *attr* element. *nodetype* can be used to set up inheritance relationships amongst the nodes using the parent *attribute*. It associates an optional shape to this type of node.

edgetype : defines the type of edge being created. Like the *nodetype* it sets up an inheritance relationship using the *parent* attribute. *edgetype* by default makes an edge directed through its Boolean directed attribute.

hostgraph: This element any combination of *attr*, *edge* and *node* elements. This element defines the graph. *node* and *edge* elements are made up of only one or more *attr* elements. These elements have some common basic definition and attributes: a required *id, type, match, label, variable* and *negative*. *match* allows morphisms to be encoded. *variable* and *negative* are both Boolean values which are false by default. Each element is a collection of attributes. The difference in these to element is the *edge* element has a *begin* attribute and an *end* attribute while a *node* element has a *xpos* attribute and a *ypos* attribute.

transformation : takes *attr* and *rewrite* elements. *rewrite* elements are made up of any number of *attr* elements , a *lhsgraph* element and a *rhsgraph* element. Both *lhsgraph* and r*hsgraph* are themselves any combination of *attr*, *node* and *edge* elements.

attr: This element comprises any number of only *attrelement* elements. It has two attributes *name* and *value*. *attrelement* element is also has the attributes *name* and *value. attr* has both a singleton attribute and a *attrelement* to allow collections.

Further details are given below in grxl.dtd.

grxl.dtd

```
<!ELEMENT grxl (attr*, nodetype*, edgetype*, hostgraph*, transformation*)>
<!ATTLIST grxl
  id ID #IMPLIED>

<!ELEMENT nodetype (attr*)>
<!ATTLIST nodetype
  id ID #REQUIRED
  parent IDREF #IMPLIED
  shape CDATA #IMPLIED
  height CDATA #IMPLIED
  width CDATA #IMPLIED>

<!ELEMENT edgetype (attr*)>
<!ATTLIST edgetype
  id ID #REQUIRED
  parent IDREF #IMPLIED
  directed (true | false) "true">

<!ELEMENT hostgraph (attr*, node*, edge*)>
<!ATTLIST hostgraph
  id ID #REQUIRED>

<!ELEMENT transformation (attr*, rewrite*)>
<!ATTLIST transformation
  id ID #REQUIRED>
```

```
<!ELEMENT rewrite (attr*, lhsgraph, rhsgraph)>
<!ATTLIST rewrite
  id ID #REQUIRED>

<!ELEMENT lhsgraph (attr*, node*, edge*)>
<!ATTLIST lhsgraph
  id ID #REQUIRED>

<!ELEMENT rhsgraph (attr*, node*, edge*)>
<!ATTLIST rhsgraph
  id ID #REQUIRED>

<!ELEMENT node (attr*)>
<!ATTLIST node
  id ID #REQUIRED
  type IDREF #IMPLIED
  match IDREF #IMPLIED
  label CDATA #IMPLIED
  xpos CDATA #IMPLIED
  ypos CDATA #IMPLIED
  variable (true | false) "false"
  negative (true | false) "false">

<!ELEMENT edge (attr*)>
<!ATTLIST edge
  id ID #REQUIRED
  type IDREF #IMPLIED
  match IDREF #IMPLIED
  begin IDREF #REQUIRED
  end IDREF #REQUIRED
  label CDATA #IMPLIED
  variable (true | false) "false"
  negative (true | false) "false">

<!ELEMENT attr (attrelement)*>
<!ATTLIST attr
  name CDATA #REQUIRED
  value CDATA #IMPLIED>

<!ELEMENT attrelement EMPTY>
<!ATTLIST attrelement
  name CDATA #REQUIRED
  value CDATA #IMPLIED>
```

AddAge.xml

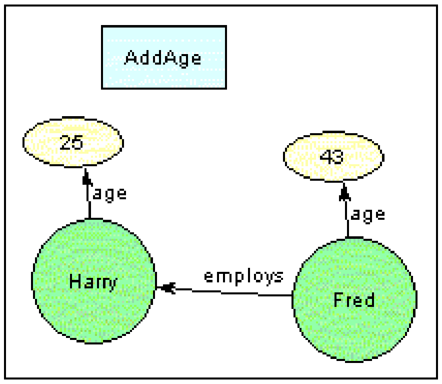


Figure**:** Host Graph

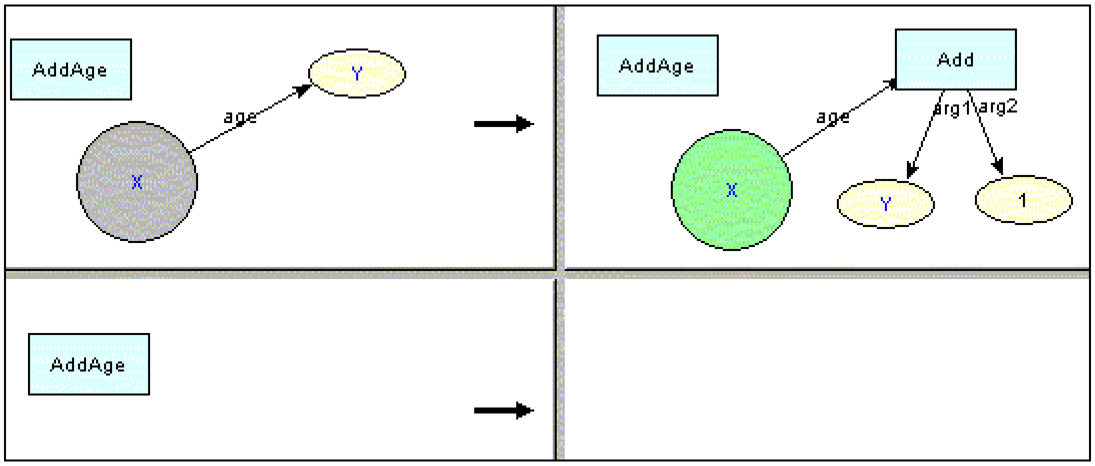
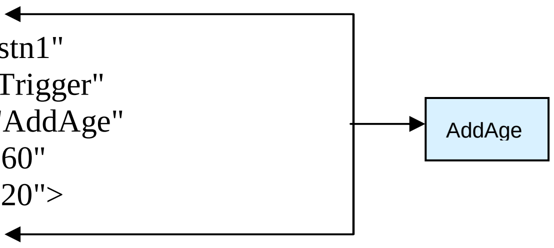


Figure: Transformation AddAge

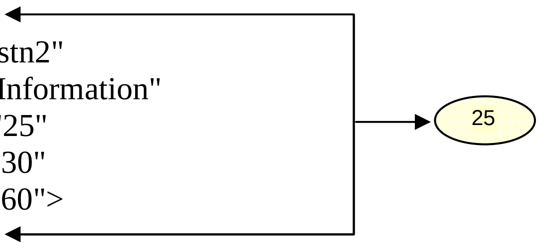
AddAge.xml

```
<?xml version = "1.0"?>
<!DOCTYPE grxl SYSTEM "grxl.dtd">
<grxl>
        <nodetype
                id = "Top">
        </nodetype>

        <nodetype
                id = "Trigger"
                parent = "Top"
                shape = "rectangle"
                height = "20"
                width = "30">
        </nodetype>
```

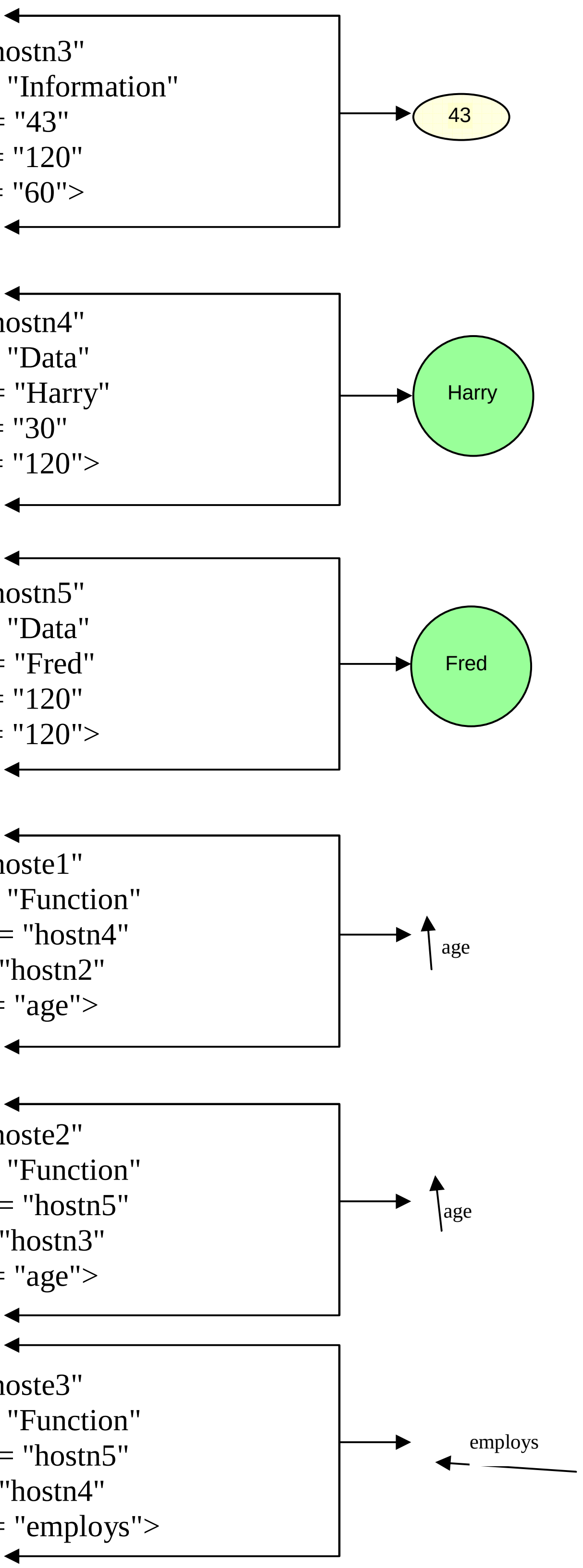
```
<nodetype
          id = "NonExecutable"
          parent = "Top">
</nodetype>

<nodetype
            id = "Data"
            parent = "NonExecutable"
            shape = "oval"
            height = "30"
            width = "30">
</nodetype>

<nodetype
            id = "Information"
            parent = "NonExecutable"
            shape = "oval"
            height = "20"
            width = "30">
</nodetype>

<edgetype
            id = "Function">
</edgetype>

<hostgraph
          id = "Start">
          <attr name = "step" value = "0"></attr>
          <node
                    id = "hostn1"
                    type = "Trigger"
                    label = "AddAge"
                    xpos = "60"
                    ypos = "20">
          </node>
```



```
          <node
                    id = "hostn2"
                    type = "Information"
                    label = "25"
                    xpos = "30"
                    ypos = "60">
          </node>
```



```
    <node
            id = "hostn3"
            type = "Information"
            label = "43"
            xpos = "120"
            ypos = "60">
    </node>
    <node
            id = "hostn4"
            type = "Data"
            label = "Harry"
            xpos = "30"
            ypos = "120">
    </node>
    <node
            id = "hostn5"
            type = "Data"
            label = "Fred"
            xpos = "120"
            ypos = "120">
    </node>
    <edge
            id = "hoste1"
            type = "Function"
            begin = "hostn4"
            end = "hostn2"
            label = "age">
    </edge>
    <edge
            id = "hoste2"
            type = "Function"
            begin = "hostn5"
            end = "hostn3"
            label = "age">
    </edge>
    <edge
            id = "hoste3"
            type = "Function"
            begin = "hostn5"
            end = "hostn4"
            label = "employs">
    </edge>
</hostgraph>
```



```
<transformation id = "AddAge">
        <rewrite id = "AddAge1">

                <lhsgraph id = "LHS1">
                        <node
                                id = "n1"
                                type = "Trigger"
                                label = "AddAge"
                                xpos = "30"
                                ypos = "30"
                                variable = "false">
                        </node>
```

Creates a new 'Trigger" node with the same label, AddAge, but positioned at (30,30)

```
                        <node
                                id = "n2"
                                type = "Data"
                                label = "X"
                                xpos = "40"
                                ypos = "60"
                                variable = "true">
                                <attr name = "once only" value = "yes"></attr>
                        </node>
```

Creates a node of type "Data' which is a variable X

```
                        <node
                                id = "n3"
                                type = "Information"
                                label = "Y"
                                xpos = "100"
                                ypos = "30"
                                variable = "true">
                        </node>
```

Creates a node of type "Information" which is a variable Y

```
                        <edge
                                id = "e1"
                                type = "Function"
                                begin = "n2"
                                end = "n3"
                                label = "age"
                                variable = "false">
                        </edge>
                </lhsgraph>
```

Creates an edge, e1, of type "Function" that points from X to Y.

```
                <rhsgraph id = "RHS1">
                        <node
                                id = "n11"
                                type = "Trigger"
```

```
        label = "AddAge"
        xpos = "30"
        ypos = "30"
        variable = "false">
</node>

<node
        id = "n12"
        type = "Data"
        label = "X"
        xpos = "40"
        ypos = "60"
        variable = "true">
        <attr name = "once only" value = "yes"></attr>
</node>

<node
        id = "n13"
        type = "Information"
        label = "Y"
        xpos = "100"
        ypos = "70"
        variable = "true">
</node>

<node
        id = "n14"
        type = "Trigger"
        label = "Add"
        xpos = "110"
        ypos = "20"
        variable = "false">
</node>

<node
        id = "n15"
        type = "Information"
        label = "1"
        xpos = "120"
        ypos = "70"
        variable = "true">
</node>

<edge
        id = "e11"
        type = "Function"
```

```
                    begin = "n12"
                    end = "n14"
                    label = "age"
                    variable = "false">
                </edge>

                <edge
                    id = "e12"
                    type = "Function"
                    begin = "n14"
                    end = "n13"
                    label = "arg1"
                    variable = "false">
                </edge>

                <edge
                    id = "e13"
                    type = "Function"
                    begin = "n14"
                    end = "n15"
                    label = "arg2"
                    variable = "false">
                </edge>
            </rhsgraph>
        </rewrite>

        <rewrite id = "AddAge2">

            <lhsgraph id = "LHS2">
                <node
                    id = "n21"
                    type = "Trigger"
                    label = "AddAge"
                    xpos = "30"
                    ypos = "30"
                    variable = "false">
                </node>
            </lhsgraph>

            <rhsgraph id = "RHS2">
            </rhsgraph>
        </rewrite>
    </transformation>
</grxl>
```

Restrictions

1. There is a huge degree of redundancy in the grammar definition
2. Addition of any extra user defined attributes can only be included at a different conceptual level by using the **attr** element
3. A very shallow approach was employed in putting so many attributes in the tags. A major deliberation in the design of this specification was depth and consistency verses shallow readability. The XML DTD grammar forces each level to be explicitly marked up. Full extensibility and generality in the DTD entails a large amount of markup for each graph transformation system stored. To overcome this, the approach adopted explicitly included the important attributes in the ATTLIST at the highest reasonable level.
4. While the grammar depth is great, the storage overhead is significant.

THE RESOURCE DESCRIPTION FRAMEWORK (RDF)

The Resource Description Framework (RDF) is a framework for representing information in the Web. RDF has an abstract syntax that reflects a simple graph-based data model, and formal semantics with a rigorously defined notion of entailment providing a basis for well founded deductions in RDF data. RDF is designed to represent information in a minimally constraining, flexible way. It can be used in isolated applications, where individually designed formats might be more direct and easily understood, but RDF's generality offers greater value from sharing. The value of information thus increases as it becomes accessible to more applications across the entire Internet.

Design Goals

- **A Simple Data Model**

RDF has a simple data model that is easy for applications to process and manipulate. The data model is independent of any specific serialization syntax.

- **Formal Semantics and Inference**

RDF has a formal semantics which provides a dependable basis for reasoning about the meaning of an RDF expression. In particular, it supports rigorously defined notions of entailment which provide a basis for defining reliable rules of inference in RDF data.

- **Extensible URI-based Vocabulary**

The vocabulary is fully extensible, being based on URIs with optional fragment identifiers (*URI references*, or *URIrefs*). URI references are used for naming all kinds of things in RDF. The other kind of value that appears in RDF data is a literal.

- **XML-based Syntax**

RDF has a recommended XML serialization form, which can be used to encode the data model for exchange of information among applications.

- **Use XML Schema Datatypes**

RDF can use values represented according to XML schema datatypes, thus assisting the exchange of information between RDF and other XML applications.

- **Anyone Can Make Statements About Any Resource**

To facilitate operation at Internet scale, RDF is an open-world framework that allows anyone to make statements about any resource.

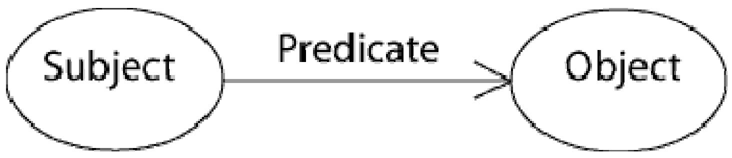
In general, it is not assumed that complete information about any resource is available. RDF does not prevent anyone from making assertions that are nonsensical or inconsistent with other statements, or the world as people see it. Designers of applications that use RDF should be aware of this and may design their applications to tolerate incomplete or inconsistent sources of information.

RDF Concepts

RDF uses the following key concepts:

- **Graph Data Model**

The underlying structure of any expression in RDF is a collection of triples, each consisting of a subject, an object and a predicate. Each triple represents a statement of a relationship between the things denoted by the nodes that it links. The predicate, also called a property, denotes the relationship. A set of such triples is called an RDF graph. This can be illustrated by a node and directed-arc diagram, in which each triple is represented as a node-arc-node link.



The direction of the arc is significant: it always points toward the object. The nodes of an RDF graph are its subjects and objects.

URI-based Vocabulary and Node Identification

A node may be a URI with optional fragment identifier (URI reference, or ***URIref***), a literal, or blank. Properties are *URI references*. A URI reference or literal used as a node identifies what that node represents. A URI reference used as a predicate identifies a relationship between the things represented by the nodes it connects. A predicate URI reference may also be a node in the graph.

Datatypes

Datatypes are used by RDF in representing values such as integers, floating point numbers and dates. A datatype consists of a lexical space, a value space and a lexical-to-value mapping. For example, the lexical-to-value mapping for the XML Schema datatype *xsd:boolean*, where each member of the value space (represented here as 'T' and 'F') has two lexical representations, is ;

Value Space	{T, F}
Lexical Space	{"0", "1", "true", "false"}
Lexical-to-Value Mapping	{<"true", T>, <"1", T>, <"0", F>, <"false", F>}

RDF predefines just one datatype ***rdf:XMLLiteral***, used for embedding XML in RDF. There is no built-in concept of numbers or dates or other common values. Rather, RDF defers to datatypes that are defined separately, and identified with URI references. The predefined XML Schema datatypes are widely used for this purpose. RDF provides no mechanism for defining new datatypes. XML Schema Datatypes provides an extensibility framework suitable for defining new datatypes for use in RDF.

Literals

Literals are used to identify values such as numbers and dates by means of a lexical representation. Anything represented by a literal could also be represented by a URI, but it is often more convenient or intuitive to use literals.A literal may be the object of an RDF statement, but not the subject or the predicate.

Literals may be *plain* or *typed* :

- A *plain literal* is a string combined with an optional language tag.
- A *typed literal* is a string combined with a datatype URI. It denotes the member of the identified datatype's value space obtained by applying the lexical-to-value mapping to the literal string.

For text that may contain markup, literals with type rdf:XMLLiteral are used.

RDF Expression of Simple Facts

Some simple facts indicate a relationship between two things. Such a fact may be represented as an RDF triple. A familiar representation of such a fact might be as a row in a table in a relational database. The table has two columns, corresponding to the subject and the object of the RDF triple. The name of the table corresponds to the predicate of the RDF triple.

Relational databases permit a table to have an arbitrary number of columns. A row expresses information corresponding to a predicate with an arbitrary number of places. Such a row, or predicate, has to be decomposed for representation as RDF triples. A simple form of decomposition introduces a new blank node, corresponding to the row, and a new triple is introduced for each cell in the row. The subject of each triple is the new blank node, the predicate corresponds to the column name, and object corresponds to the value in the cell. The new blank node may also have an rdf:type property whose value corresponds to the table name.

As an example, consider Figure :

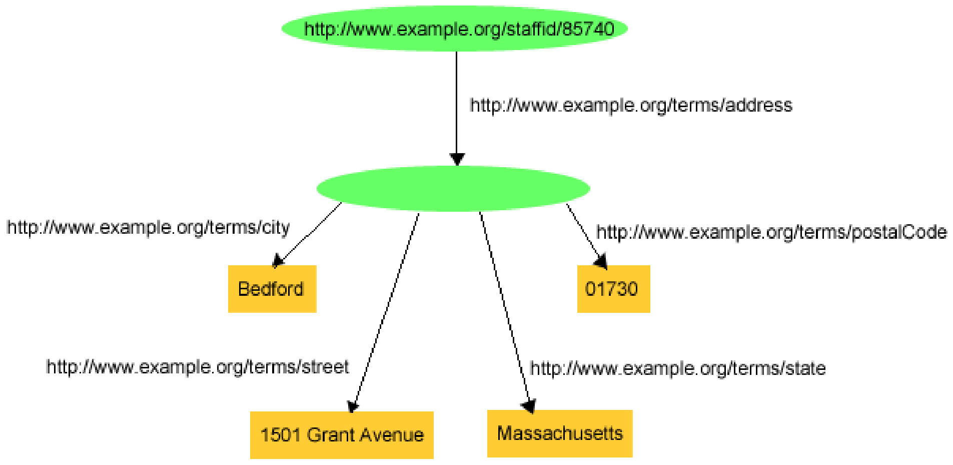

http://www.example.org/staffid/85740
http://www.example.org/terms/address
http://www.example.org/terms/city
Bedford
http://www.example.org/terms/postalCode
01730
http://www.example.org/terms/street
1501 Grant Avenue
http://www.example.org/terms/state
Massachusetts

Figure: Using a Blank Node

This information might correspond to a row in a table "STAFFADDRESSES", with a primary key STAFFID, and additional columns STREET, STATE, CITY and POSTALCODE.

Thus, a more complex fact is expressed in RDF using a conjunction (logical-AND) of simple binary relationships. RDF does not provide means to express negation (NOT) or disjunction (OR).

Through its use of extensible URI-based vocabularies, RDF provides for expression of facts about arbitrary subjects; that is assertions of named properties about specific named things. A URI can be constructed for any thing that can be named, so RDF facts can be about any such things.

Entailment

The ideas on meaning and inference in RDF are underpinned by the formal concept of *entailment*. In brief, an RDF expression A is said to ***entail*** another RDF expression B if every possible arrangement of things in the world that makes A true also makes B true. On this basis, if the truth of A is presumed or demonstrated then the truth of B can be inferred .

An XML Syntax for RDF: RDF/XML

RDF provides XML syntax for writing down and exchanging RDF graphs, called *RDF/XML*. Unlike triples, which are intended as a shorthand notation, RDF/XML is the normative syntax for writing RDF.

The basic ideas behind the RDF/XML are as follows:

Consider the English statement:

http://www.example.org/index.html has a creation-date whose value is August 16, 1999

The RDF graph for this statement, after assigning a URIref to the creation-date property, is:

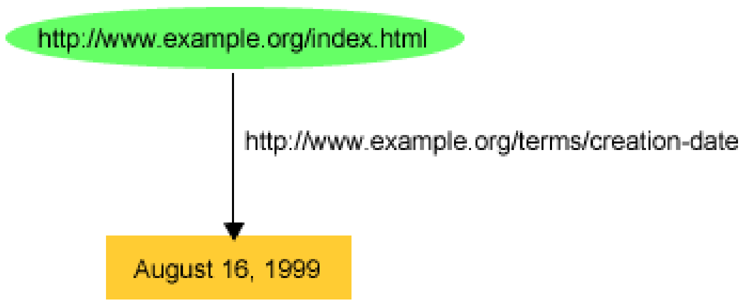

http://www.example.org/index.html
http://www.example.org/terms/creation-date
August 16, 1999

Figure : Describing a Web Page's Creation Date

The triple representation is:

ex:index.html   exterms:creation-date   "August 16, 1999" .

The RDF/XML syntax corresponding to the graph above is:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
            xmlns:exterms="http://www.example.org/terms/">
  <rdf:Description rdf:about="http://www.example.org/index.html">
      <exterms:creation-date>August 16, 1999</exterms:creation-date>
  </rdf:Description>

</rdf:RDF>
```

This seems like a lot of overhead.

This illustrates the basic ideas used by RDF/XML to encode an RDF graph as XML elements, attributes, element content, and attribute values.

Line 2 begins an rdf:RDF element which indicates that the following XML content is intended to represent RDF. Following this is the XML namespace declaration.

Lines 4-6 provide the RDF/XML for the specific statement shown. RDF/XML represents the statement *by considering the statement as a description*, and that it is *about* the subject of the statement. The rdf:Description start-tag in line 4 indicates the start of a *description* of a resource, and goes on to identify the resource the statement is *about* (the subject of the statement) using the rdf:about attribute to specify the URIref of the subject resource.

Line 5 provides a *property element,* with the QName exterms:creation-date as its tag, to represent the predicate and object of the statement. The QName exterms:creation-date is chosen so that appending the local name creation-date to the URIref of the exterms: prefix (http://www.example.org/terms/) gives the statement's predicate URIref http://www.example.org/terms/creation-date. The content of this property element is the object of the statement, the plain literal August 19, 1999 that is, the value of the creation-date property of the subject resource). The property element is nested within the containing rdf:Description element, indicating that this property applies to the resource specified in the rdf:about attribute of the rdf:Description element.

Using an rdf:RDF element to enclose RDF/XML content is optional in situations where the XML can be identified as RDF/XML by context. However, it does not hurt to provide the rdf:RDF element in any case.

An RDF graph consisting of multiple statements can be represented in RDF/XML by using RDF/XML to separately represent each statement. For example, to write the following two statements:

```
ex:index.html   exterms:creation-date   "August 16, 1999" .

ex:index.html   dc:language             "en" .
```

An arbitrary number of additional statements could be written in the same way, using a separate rdf:Description element for each additional statement.

The RDF/XML syntax provides a number of abbreviations to make common uses easier to write. For example, it is typical for the same resource to be described with several properties and values at the same time. To handle such cases, RDF/XML allows multiple property elements representing those properties to be nested within the rdf:Description element that identifies the subject resource. For example, to represent the following group of statements about http://www.example.org/index.html:

```
ex:index.html   dc:creator              exstaff:85740 .
ex:index.html   exterms:creation-date   "August 16, 1999" .
ex:index.html   dc:language             "en" .
```

RDF/XML can also represent graphs that include nodes that have no URIrefs, i.e., the *blank nodes*.

GraphXML

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.

GRAPH STRUCTURES

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>
```

The first line is required in all XML files. The second line identifies the file’s type, that is 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 in the <GraphXML> tags. The real 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 defines two nodes and a connecting edge. 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.

“Attributes” can also be defined for each of the elements: these are key–value pairs. The DTD of GraphXML defines, for each element, the set of allowable attributes. It is partly through those attributes that additional information about nodes, edges, or graphs can be conveyed to the application.

DETAILED SPECIFICATION OF GRAPHXML

The root tag: GraphXML

A GraphXML file can consist of several graph specifications

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

The graph element

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 >
```

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.

```
<!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. If the node is a metanode, the *<subgraphstyle>* 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. 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. 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).

The edge element

An *edge* tag 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 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

Geometry specifications

GraphXML can store the geometric positions of the nodes and the edges. Once this information is available, the visualization system can 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. Geometry features stored by GraphXML are:

- Node positions: Add the <position> tag as a child to <node>.
- Edge positions: Use the <path> tag that contains a sequence of control points.
- Graph size: The <size> tag can also be used as a direct child element of <graph> to denote the full size, or bounding box, of the full graph.
- Geometry for Hierarchical graphs: Use the <transform> tag, which is a child of <node>. This element has no meaning if the node is not a metanode.

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.

Specification of application 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 associated with a graph, a node, or an edge. GraphXML defines the following application data:

- **Labels (<label> tag):** 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.

Visual attribute control

Beyond the pure geometry, the visual appearance of a graph is also determined by visual properties, such as line, width, colour of the components and icons replacing nodes. In GraphXML, 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. 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.

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

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 that 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>
```

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:

1. Adding properties to tags and
2. Adding new tag specification to existing tags.

TRANSFORMING GRAPHXML FILES

The primary goal of GraphXML is to describe graphs that are to be visualized by specialized applications. However, one can use the XSLT mechanism 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.

Restrictions

The user extension mechanism of GraphXML is based on the use of internal DTD's, which do not have a very friendly syntax.

GraphML

The purpose of a GraphML document is to define a graph. The GraphML document consists of a *graphml* element and a variety of sub elements: *graph*, *node* and *edge*. The first line of the document is an XML process instruction that defines that the document adheres to the XML 1.0 standard and that the encoding of the document is UTF-8, the standard encoding for XML documents. Of course other encodings can be chosen for GraphML documents.

Common Elements

The part of the document that is common to all GraphML documents is basically the *graphml* element.

The *root-element* element of a GraphML document is the *graphml* element. The *graphml* element, like all other GraphML elements, belongs to the namespace *http://graphml.graphdrawing.org/xmlns*. This namespace is defined as the default namespace in the document by adding the XML Attribute *xmlns="http://graphml.graphdrawing.org/xmlns"* to it. The two other XML Attributes are needed to specify the XML Schema for this document.

Graph

A graph is, not surprisingly, denoted by a *graph* element. Nested inside a *graph* element are the declarations of nodes and edges. A node is declared with a *node* element, and an edge with an *edge* element. In GraphML there is no order defined for the appearance of *node* and *edge* elements.

Graphs in GraphML are *mixed*, in other words, they can contain directed and undirected edges at the same time. If no direction is specified when an edge is declared, the *default direction* is applied to the edge. The default direction is declared as the XML Attribute *edgedefault* of the *graph* element. The two possible value for this XML Attribute are *directed* and *undirected*. Note that the default direction *must* be specified.

Optionally an identifier *id* can be specified and used, when it is necessary to reference the graph.

Node

Nodes in the graph are declared by the *node* element. Each node has an identifier, which must be unique within the entire document, that is, in a document there must be no two nodes with the same identifier. The identifier of a node is defined by the XML-Attribute *id*.

Edge

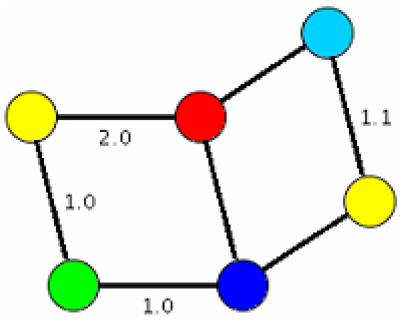
Edges in the graph are declared by the *edge* element. Each edge must define its two endpoints with the XML-Attributes *source* and *target*.

Edges with only one endpoint; also called loops, self loops, or reflexive edges; are defined by having the same value for *source* and *target*. The optional XML-Attribute *directed* declares if the edge is directed or undirected. The value *true* declares a directed edge, the value *false* an undirected edge. Optionally the identifier *id* can be specified and used, when it is necessary to reference the edge.

GraphML-Attributes

While pure topological information may be sufficient for some applications of GraphML, for the most time additional information is needed. With the help of the extension *GraphML-Attributes* one can specify additional information of simple type (scalar) for the elements of the graph.

To add structured content to graph elements the key/data extension mechanism of GraphML is used. *GraphML-Attributes* are used to store the extra data on the nodes and edges. The following example illustrates.



A graph with colored nodes and edge weights.

Example of a GraphML Document with GraphML-Attributes

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
    http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
        <key id="d0" for="node" attr.name="color" attr.type="string">
                <default>yellow</default>
        </key>
        <key id="d1" for="edge" attr.name="weight" attr.type="double"/>
        <graph id="G" edgedefault="undirected">
                <node id="n0">
                        <data key="d0">green</data>
                </node>
                <node id="n1"/>
                <node id="n2">
                        <data key="d0">blue</data>
                </node>
                <node id="n3">
                        <data key="d0">red</data>
                </node>
                <node id="n4"/>
				<node id="n5">
					<data key="d0">turquoise</data>
				</node>
				<edge id="e0" source="n0" target="n2">
					<data key="d1">1.0</data>
				</edge>
				<edge id="e1" source="n0" target="n1">
					<data key="d1">1.0</data>
				</edge>
					<data key="d1">2.0</data>
				</edge>
				<edge id="e3" source="n3" target="n2"/>
				<edge id="e4" source="n2" target="n4"/>
				<edge id="e5" source="n3" target="n5"/>
				<edge id="e6" source="n5" target="n4">
					<data key="d1">1.1</data>
				</edge>
		</graph>
</graphml>
```

A *GraphML-Attribute* is defined by a *key* element that specifies the *identifier*, *name*, *type* and *domain* of the attribute.

The *identifier* is specified by the XML-Attribute *id* and is used to refer to the *GraphML-Attribute* inside the document.

The *name* of the *GraphML-Attribute* is defined by the XML-Attribute *attr.name* and must be unique amongst all *GraphML-Attributes* declared in the document. The purpose of the name is so that applications can identify the meaning of the attribute. The name of the GraphML-Attribute is not used inside the document; the identifier is used for this purpose.

The *GraphML-Attribute* can be of *type*: *boolean, int, long, float, double,* or *string.*

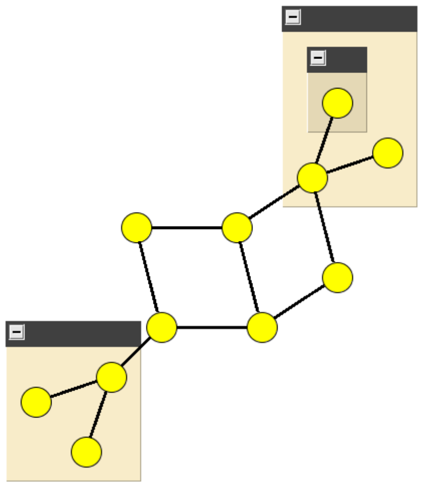
The domain of the GraphML-Attribute specifies for which graph elements the GraphML-Attribute is declared. Possible values include graph, node, edge, and all.

The value of a GraphML-Attribute for a graph element is defined by a data element nested inside the element for the graph element. The data element has an XML-Attribute key, which refers to the identifier of the *GraphML-Attribute*. The value of the *GraphML-Attribute* is the text content of the data element. This value must be of the type declared in the corresponding key definition.

There can be graph elements for which a *GraphML-Attribute* is defined but no value is declared by a corresponding data element. If a default value is defined for this GraphML-Attribute, then this default value is applied to the graph element. If no default value is specified, the value of the GraphML-Attribute is undefined for the graph element.

Nested Graphs

GraphML supports nested graphs, that is, graphs in which the nodes are hierarchically ordered. The hierarchy is expressed by the structure of the GraphML document. A node in a GraphML document may have a graph element which itself contains the nodes which are in the hierarchy below this node. In drawing the graph the hierarchy is expressed by containment, that is, the a node a is below a node b in the hierarchy if and only if the graphical representation of a is entirely inside the graphical representation of b. An example for a nested graph and the corresponding GraphML document is given below.



A nested graph.

GraphML Document with Nested Graphs

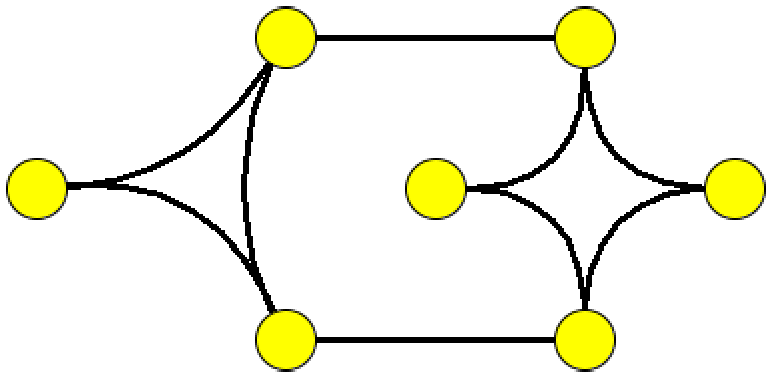
```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
        http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">

            <graph id="G" edgedefault="undirected">
                        <node id="n0"/>
                        <node id="n1"/>
                        <node id="n2"/>
                        <node id="n3"/>
                        <node id="n4"/>
                        <node id="n5">
                                    <graph id="n5:" edgedefault="undirected">
                                                <node id="n5::n0"/>
                                                <node id="n5::n1"/>
                                                <node id="n5::n2"/>
                                                <edge id="e0" source="n5::n0" target="n5::n2"/>
                                                <edge id="e1" source="n5::n1" target="n5::n2"/>
                                    </graph>
                        </node>
                        <node id="n6">
                                    <graph id="n6:" edgedefault="undirected">
                                                 <node id="n6::n0">
                                                            <graph id="n6::n0:" edgedefault="undirected">
                                                                        <node id="n6::n0::n0"/>
                                                            </graph>
                                                </node>
                                                <node id="n6::n1"/>
                                                <node id="n6::n2"/>
                                                <edge id="e10" source="n6::n1" target="n6::n0::n0"/>
                                                <edge id="e11" source="n6::n1" target="n6::n2"/>
                                    </graph>
                        </node>
                        <edge id="e2" source="n5::n2" target="n0"/>
                        <edge id="e3" source="n0" target="n2"/>
                        <edge id="e4" source="n0" target="n1"/>
                        <edge id="e5" source="n1" target="n3"/>
                        <edge id="e6" source="n3" target="n2"/>
                        <edge id="e7" source="n2" target="n4"/>
                        <edge id="e8" source="n3" target="n6::n1"/>
                        <edge id="e9" source="n6::n1" target="n4"/>
            </graph>
</graphml>
```

The edges between two nodes in a nested graph have to be declared in a graph, which is an ancestor of both nodes in the hierarchy. In the example, declaring the edge between node *n6::n1* and node *n4::n0::n0* inside graph *n6::n0* would be wrong while declaring it in graph *G* would be correct. A good policy is to place the edges at the least common ancestor of the nodes in the hierarchy, or at the top level.

For applications that cannot handle nested graphs the fallback behaviour is to ignore nodes that are not contained in the top-level graph and to ignore edges that do not have both endpoints in the top-level graph.

Hyperedges

Hyperedges are a generalization of edges in the sense that they do not only relate two endpoints to each other, they express a relation between an arbitrary number of endpoints. Hyperedges are declared by a *hyperedge* element in GraphML. For each endpoint of the hyperedge, this *hyperedge* element contains an *endpoint* element. The *endpoint* element must have an XML-Attribute *node*, which contains the identifier of a node in the document. The hyperedges are illustrated by joining arcs, the edges, by straight lines. Edges can be either specified by an *edge* element or by a *hyperedge* element containing two *endpoint* elements. The following example contains two hyperedges and two edges.



A graph with hyperedges.

GraphML Document with Hyperedges

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
           http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
     <graph id="G" edgedefault="undirected">
            <node id="n0"/>
            <node id="n1"/>
            <node id="n2"/>
            <node id="n3"/>
            <node id="n4"/>
            <node id="n5"/>
            <node id="n6"/>
            <hyperedge>
                        <endpoint node="n0"/>
                        <endpoint node="n1"/>
                        <endpoint node="n2"/>
            </hyperedge>
            <hyperedge>
                        <endpoint node="n3"/>
                        <endpoint node="n4"/>
                        <endpoint node="n5"/>
                        <endpoint node="n6"/>
            </hyperedge>
            <hyperedge>
                        <endpoint node="n1"/>
                        <endpoint node="n3"/>
            </hyperedge>
            <edge source="n0" target="n4"/>
     </graph>
   </graphml>
```

Like edges, hyperedges and endpoints may have an XML-Attribute *id*.

Ports

A node may specify different logical locations for edges and hyperedges to connect. The logical locations are called "ports".

The ports of a node are declared by *port* elements as children of the corresponding *node* elements. Note that port elements may be nested, that is, they may contain *port* elements themselves. Each port element must have an XML-Attribute *name*, which is an identifier for this port. The *edge* element has optional XML-Attributes *sourceport* and *targetport* with which an edge may specify the port on the *source* or *target* node. Correspondingly, the *endpoint* element has an optional XML-Attribute *port*.

Extending GraphML

GraphML is designed to be easily extensible. With GraphML the topology of a graph and simple attributes of graph elements can be serialized. To store more complex application data one has to extend GraphML. An XML Schema defines extensions of GraphML. The Schema which defines the extension can be derived from the GraphML Schema documents by using a standard mechanism similar to the to one used by XHTML.

Adding XML Attributes to GraphML Elements

In most cases, additional information can be attached to GraphML elements by usage of *GraphML-Attributes*, this assures readability for other GraphML parsers. However, sometimes it might be more convenient to use XML attributes.

GRAPH EXCHANGE LANGUAGE – GXL

GXL is designed to be a standard software exchange format for exchanging information derived from programs, and more generally for exchanging information that is conveniently represented as a graph. To facilitate this exchange the information is represented as a graph and the graph is transcribed the to XML.

Being a structural description, graphs have no meaning of their own. The meaning of graphs corresponds to the context in which they are exchanged. This context is, in part, given by a schema (essentially an E/R diagram or class diagram). In GXL, both the actual data representing the graph and the schema are passed as XML stream. Other information associated with the graph, such as user annotations or (x, y) locations for graph layout, is attached to the graph and passed in GXL as attributes.

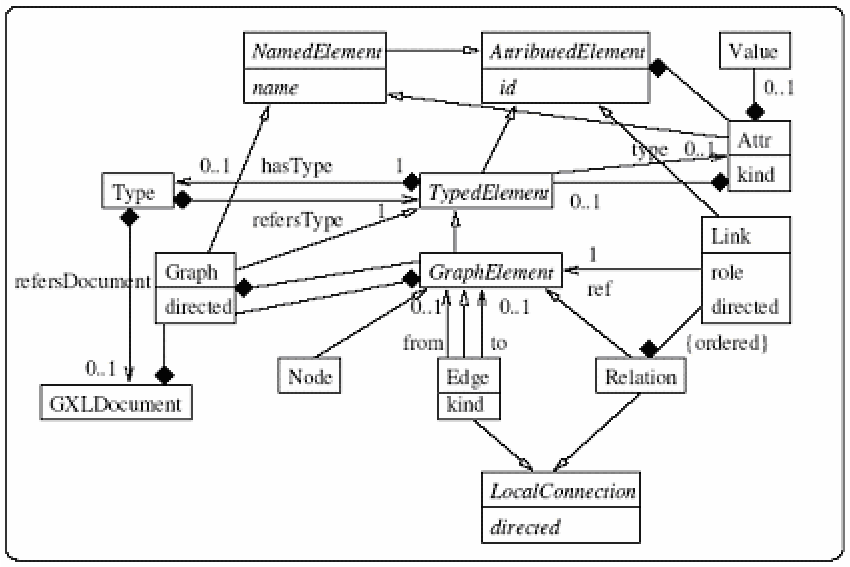
GXL is an XML sublanguage that offers an adaptable and flexible means to support interoperability between graph-based tools. GXL allows the combination of single purpose tools especially parsers, source code extractors, source code visualizers, into a single powerful workbench. Since GXL is a general graph exchange format, it can also be used to interchange any graph-based data including models between CASE tools, data between graph transformations systems and graph visualization tools. GXL includes support for hypergraphs and hierarchical graphs and can be extended to support other types of graphs.

The Logical Graph Model

A GXL Document contains a set of *Graph*s where each has a set of *GraphElement*s. A graph element can be a *Node*, an *Edge* or a *Relation*, able to store hyperedges. *Edges* and *Relations* can both be directed, expressed by their common super class *LocalConnection*. Moreover, partial graphs with dangling edges are allowed, since the multiplicities of associations from and to are allowed to be 0 and 1. *Relations* have *Links* storing role names and directions and pointing to graph elements. In this way edges on edges or hyperedges are possible. The design of GXL is such that edges are distinguished from relations in order to increase the readability of XML documents for simple graphs. However, edges may be seen as special relations with two links 'from' and 'to'.

*Graph*s and *GraphElement*s are *TypedElement*s meaning that there are type graphs where the graph elements function as types for *Nodes*, *Edges* and *Relations*. It is also possible that a type graph is stored in another GXL document, thus it has to keep a corresponding pointer (Type).

Each *GraphElement* is an *AttributedElement* that can have a set of attributes. Each attribute has a *name*, a *type* and a *value*. The *value* can be a primitive value, a container or a complex value that might be located in another document. See Figure .

Syntax of 

Logical model for graphs **GXL**

XML documents are trees and storing graphs in trees means linearizing their two-dimensional structure. Such a translation is more or less automatic. DTD (Document Type Definition) notation is used to specify the syntax of GXL. In the DTD corresponding to the model Figure each concrete class of the model results in an element. The class members and associations define the attribute list (ATTLIST) of this element. The associations produce attribute entries pointing to some ID of another element, being an IDREF. The type of such a reference cannot be expressed in a DTD; therefore an additional comment explains which types are expected. Aggregations are translated to sub element relations. Note that in contrast to the logical model sub elements can be ordered in a DTD. Abstract classes are not translated to elements, but their members, associations and aggregations are added as attributes and sub elements to all elements generated from inheriting classes.

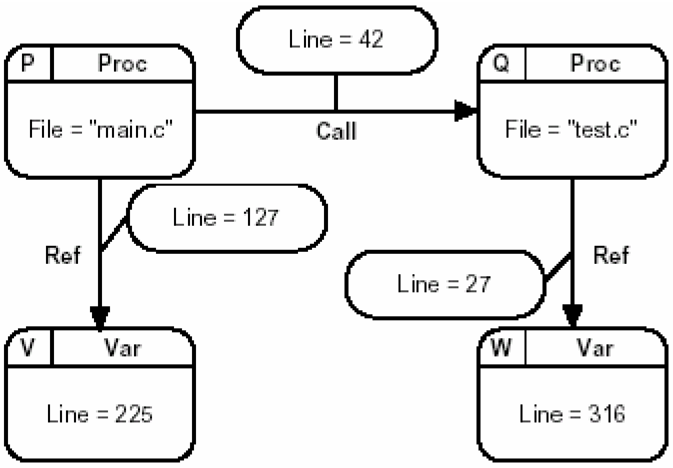
```
<!ELEMENT GXLDocument (Graph*)>

<!ELEMENT Graph (Attr*, (Node | Edge | Relation)*)>
<!ATTLIST Graph
        id ID #REQUIRED
        name NMTOKEN #IMPLIED
        type IDREF #IMPLIED
        directed (true | false) "true">

<!ELEMENT Node (Attr*, Graph* )>
<!ATTLIST Node
        id ID #REQUIRED
        type IDREF #IMPLIED>

<!ELEMENT Edge (Attr*, Graph*)>
<!ATTLIST Edge
        id ID #IMPLIED
        type IDREF #IMPLIED <!-- Type -->
        from IDREF #IMPLIED
        to IDREF #IMPLIED
        kind (refine|normal) #IMPLIED
        directed (true|false) #IMPLIED>

<!ELEMENT Relation (Attr*, (Link| Graph)* )>
<!ATTLIST Relation
        id ID #IMPLIED
        type IDREF #IMPLIED
        directed (true|false) #IMPLIED>

<!ELEMENT Link (Attr*)>
<!ATTLIST Link
        ref IDREF #REQUIRED
        role NMTOKEN #IMPLIED
        directed (true|false) #IMPLIED>

<!ELEMENT Attr (Attr*, Value? )>
<!ATTLIST Attr
        name NMTOKEN #REQUIRED
        type IDREF #IMPLIED
        kind NMTOKEN #IMPLIED>
```

Figure: GXL DTD

Data as Typed Graphs

Figure shows a graph that represents a fragment on a program. The graph shows the various calls, reference variables, file location of procedures, source lines for variable declarations and the source lines in the program where the calls and references in the program occur (lines 127, 42 and 27).

It is common to represent data about software as diagrams similar to Figure . Such a diagram is an attributed (*File* and *Line* are attributes), typed (*Proc* and *Var* are types), directed graph, or simply a *typed graph* for short. This mathematical model (typed graphs) provides a clear meaning of the data being exchanged. The means of encoding the data as a stream of bytes is not considered though.



Sample typed graph with attributes

The XML stream below represents the sample type graph with attributes shown above.

```
<gxl>
        <node id="P" type="Proc">
                <attr name="File" value="main.c" />
        </node>

        <node id="Q" type="Proc">
                <attr name="File" value="test.c" />
        </node>

        <node id="V" type="Var">
                <attr name="Line" value="225" />
        </node>

        <node id="W" type="Var">
                <attr name="Line" value="316" />
        </node>

        <edge begin="P" end="Q" type="Call">
                <attr name="Line" value="42" />
        </edge>

        <edge begin="P" end="V" type="Ref">
                <attr name="Line" value="127" />
        </edge>

        <edge begin="Q" end="W" type="Ref">
                <attr name="Line" value="316" />
        </edge>
</gxl>
```

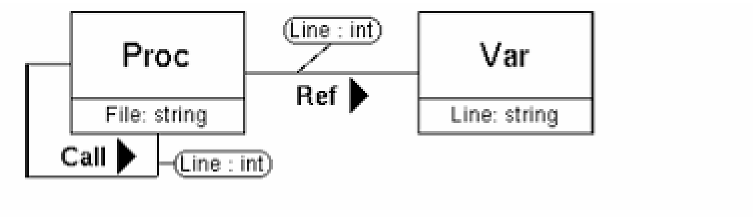
Figure Graph in Figure represented in XML (as an GXL document). The nodes (*P*, *Q*, *V* and *W*) and edges (*P*,*Q*), (*P*, *V*) and (*Q*, *W*) are represented along with their types and attributes

Since software is written in various programming languages and the level of granularity varies, it is inevitable that the types and attributes in typed graphs vary according to purpose. GXL deals with this variability in the same manner in which relational databases deal with it; by means of schemas that specify the form of the graph data. In the case of facts about software, schemas are quite changeable as opposed to classical databases, which have rarely changing schemas.

A common way to specify the schema for a typed graph is by means of (Extended) E/R (Entity/Relation) diagrams or UML class diagrams. Figure gives an UML diagram, or schema, that specifies the general form of the graphs like the one in Figure. This figure specifies that

Proc’s can call *Proc*’s and can reference *Var*’s. It specifies that *Proc*’s has a *File* attribute, *Var*’s have a *Line* attribute and that *Call* and *Ref* edges have a *Line* attribute.

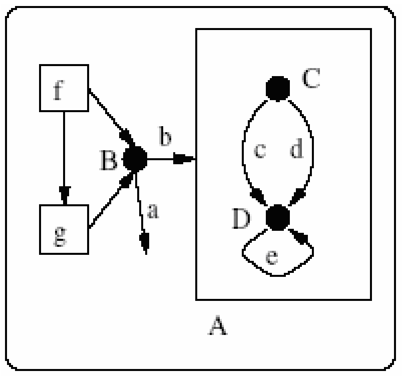
Since class diagrams represent structured information, they can be represented as a typed graph as well. By representing schemata by graphs, graphs can be directly represented in GXL, as XML streams. This allows transmission of the schema in XML along with the graph itself from one tool to another. Resultantly tools become independent of structural details such as the specific types and attributes in a graph. This yields tools that are parameterized by the schema.



Schema (class diagram) for graphs of the form of the one in Figure

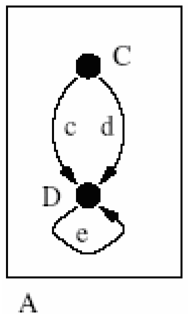
A Detailed Sample GXL Graph

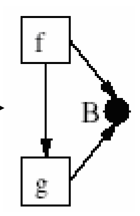
A sample graph is given which contains nearly all the structural features a graph can have. Node A is refined to the graph part drawn inside of this node. There are relations (hyperedges) f and g where one link of f points to g. Edge a is partial, it does not have a target. Edges c and d are parallel edges between the same nodes C and D and edge e is a loop. Edge b runs between a simple and a compound node. The GXL document corresponding to the graph below and to the DTD given above is shown below.



A sample graph

```
<gxl>
        <Graph id="G">
                <Node id="A">
                        <Graph id = "sub-A">
                                <Node id="C"> </Node>
                                <Node id="D"> </Node>
                                <Edge id="c" from="C" to="D"> </Edge>
                                <Edge id="d" from="C" to="D"> </Edge>
                                <Edge id="e" from="D" to="D"> </Edge>
                        </Graph>
                </Node>

                <Node id="B"> </Node>

                <Edge id="a" from="B"> </Edge>

                <Edge id="b" from="B" to="A"> </Edge>

                <Relation id="f">
                        <Link ref="B"> </Link>
                        <Link ref="g"> </Link>
                </Relation>

                <Relation id="g">
                        <Link ref="B"> </Link>
                </Relation>
        </Graph>
</gxl>
```


C
c
d
D
e
A


f
B
g