GraphML Progress Report*

Structural Layer Proposal

Ulrik Brandes[1], Markus Eiglsperger[2], Ivan Herman[3], Michael Himsolt[4], and M. Scott Marshall[3]

[1] Department of Computer & Information Science, University of Konstanz.
Ulrik.Brandes@uni-konstanz.de
[2] Wilhelm Schickard Institute for Computer Science, University of Tübingen.
eiglsper@informatik.uni-tuebingen.de
[3] Centrum voor Wiskunde en Informatica. {ivan|scott}@cwi.nl
[4] DaimlerChrysler Research. Michael.Himsolt@daimlerchrysler.com

Abstract. Following a workshop on graph data formats held with the 8th Symposium on Graph Drawing (GD 2000), a task group was formed to propose a format for graphs and graph drawings that meets current and projected requirements.
On behalf of this task group, we here present GraphML (Graph Markup Language), an XML format for graph structures, as an initial step towards this goal. Its main characteristic is a unique mechanism that allows to define extension modules for additional data, such as graph drawing information or data specific to a particular application. These modules can freely be combined or stripped without affecting the graph structure, so that information can be added (or omitted) in a well-defined way.

1 Introduction

Graph drawing tools, like all other tools dealing with relational data, need to store and exchange graphs and associated data. Despite several earlier attempts to define a standard, no agreed-upon format is widely accepted and, indeed, many tools support only a limited number of custom formats which are typically restricted in their expressibility and specific to an area of application.

Motivated by the goals of tool interoperability, access to benchmark data sets, and data exchange over the Web, the Steering Committee of the Graph Drawing Symposium started a new initiative with an informal workshop held in conjunction with the 8th Symposium on Graph Drawing (GD 2000) [1]. As a consequence, an informal task group was formed to propose a modern graph exchange format suitable in particular for data transfer between graph drawing tools and other applications.

On behalf of this group we propose GraphML (Graph Markup Language), an XML format that takes a unique approach to represent graphs and graph drawings by specifying

* The latest information on GraphML is maintained on the GraphML homepage [2].

P. Mutzel, M. Jünger, and S. Leipert (Eds.): GD 2001, LNCS 2265, pp. 501–512, 2002.

1. core elements to describe graph structures together with
2. an extension mechanism that allows to independently build application-specific graph data formats on top of them.

In particular, such extensions can be freely combined or ignored without affecting the graph data itself. Thus, drawing information can be added to an application-specific format, and graphs can be extracted from foreign application data. These features seem to be essential requirements for today's and future graph data formats, since graph models are ubiquitous and there will certainly be no agreement on a single general format across all disciplines.

This report is organized as follows. In Sect. 2, we outline the guidelines used in the design of GraphML. The core of the language is described in Sect. 3 and in Sect. 4 we outline how to add non-structural data and thus bind GraphML to specific applications. We conclude with future plans in Sect. 5.

2 Usage Scenarios and Design Goals

A modern graph exchange format cannot be defined in a monolithic way, since graph drawing services are used as components in larger systems and Web-based services are emerging. Graph data may need to be exchanged between such services, or stages of a service, and between graph drawing services and systems specific to areas of applications.

The typical usage scenarios that we envision for the format are centered around systems designed for arbitrary applications dealing with graphs and other data associated with them. Such systems will contain or call graph drawing services that add or modify layout and graphics information. Moreover, such services may compute only partial information or intermediate representations, for instance because they instantiate only part of a staged layout approach such as the topology-shape-metrics or Sugiyama frameworks. We hence aimed to satisfy the following key goal.

> The graph exchange format should be able to represent arbitrary graphs with arbitrary additional data, including layout and graphics information. The additional data should be stored in a format appropriate for the specific application, but should not complicate or interfere with the representation of data from other applications.

GraphML is designed with this and the following more pragmatic goals in mind:

- *Simplicity*: The format should be easy to parse and interpret for both humans and machines. As a general principle, there should be no ambiguities and thus a single well-defined interpretation for each valid GraphML document.
- *Generality*: There should be no limitation with respect to the graph model, i.e. hypergraphs, hierarchical graphs, etc. should be expressible within the same basic format.

– *Extensibility*: It should be possible to extend the format in a well-defined way to represent additional data required by arbitrary applications or more sophisticated use (e.g., sending a layout algorithm together with the graph).
– *Robustness*: Systems not capable of handling the full range of graph models or added information should be able to easily recognize and extract the subset they can handle.

There was no arguing that the format be based on XML (eXtensible Markup Language) [7] to stay compatible with other emerging standards such as, e.g., SOAP (Simple Object Access Protocol) [8], but also to enable use of the many widely supported tools for parsing and handling XML-formatted data. Another principal decision was to conceptually separate different layers of information, such as graph structure, application data, topology, geometry, or graphics. Figure 1 sketches the conceptual units of our design.

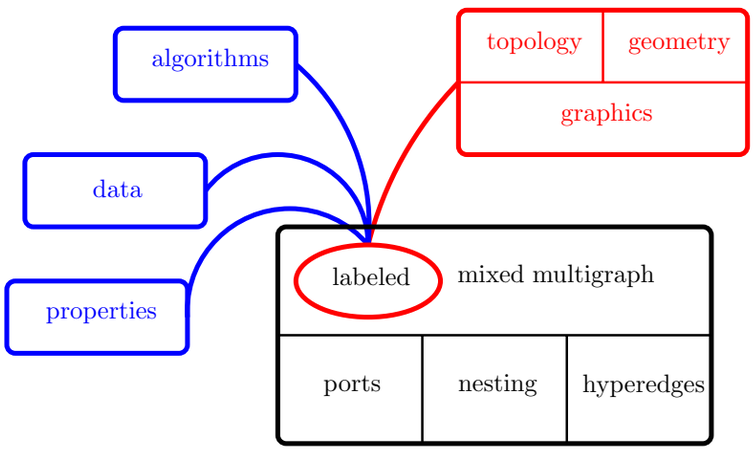


Fig. 1. The basic graph model of GraphML are labeled mixed multigraphs with optional node ports, hyperedges, and nesting. Graph drawing information is planned to be separated into topological and geometric, with a graphics layer on top. Like any other associated data, it will be encapsulated in a special tag

To date, the GraphML group has specified the structural layer, i.e. the core elements of the format describing the incidence structure of arbitrary graphs, and an extension mechanism to add non-structural data. Using this extension mechanism, it is possible to define application-specific modules that can be added to the structural layer common to all variants thus created. Information relevant to graph drawing services will later be defined within one or more such modules.

3 Structural Layer

In this section, we describe how graphs are represented in GraphML. This part of GraphML is called the structural layer and constitutes the essence of the format. The fundamental graph model underlying our design is a labeled mixed multigraph, which may, but need not, include node ports, hyperedges, and nested graphs. When these concepts are not supported by an application, they can be easily identified and may simply be ignored without losing the remaining structural information.

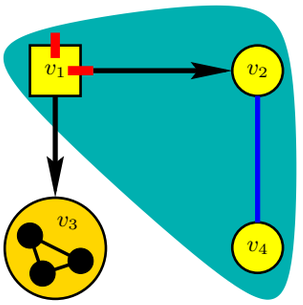


Fig. 2. Example graph in which the shaded area represents a hyperedge with three incident vertices

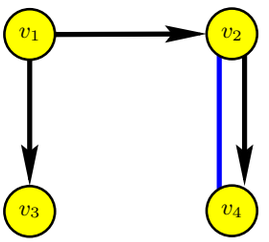
We describe separately the XML constructs defined to represent mixed multigraphs, ports, hyperedges and nesting, and use the graph in Fig. 2 as our running example. How to define and include additional data is explained in the next section, and the actual DTD (Document Type Definition) of GraphML is given in the appendix.

3.1 Mixed Multigraphs

A *mixed multigraph* is a graph which may contain both directed and undirected edges and may have loops and multi-edges. The representation chosen for GraphML is a simple list if nodes and edges. GraphML defines XML tags `<graph>`, `<node>`, and `<edge>` for this purpose.

- A GraphML document may contain any number of `<graph>`s.
- The mandatory XML attribute `edgedefault` of `<graph>` specifies whether `<edge>`s are directed or undirected by default. The optional XML attribute `directed` of `<edge>` can be used to overwrite the default.
- An `<edge>` refers to a source and a target node, regardless of whether it is directed or not.
- A `<graph>` tag may contain `<node>`s and `<edge>`s in any order.

Tools supporting only mixed multigraphs may describe and view the example graph as shown in Fig. 3. Note, however, that they may also choose to represent the features they do not support in a different way (e.g., by adding a dummy node for each hyperedge). Although there is a single well-defined interpretation for each GraphML document, no prescriptions are made on the representation of more complex graph models when only basic elements are supported.

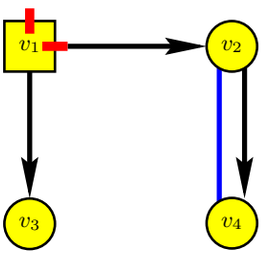



```
<graph edgedefault="directed">
  <desc>This is a mixed multigraph</desc>

  <node id="v1"/>
  <node id="v2"/>
  <node id="v3"/>
  <node id="v4"/>

  <edge source="v1" target="v2"/>
  <edge source="v1" target="v3"/>
  <edge source="v2" target="v4"/>
  <edge source="v2" target="v4" directed="false"/>
</graph>
```

Fig. 3. The graph of Fig. 2 as represented in the most basic layer of GraphML

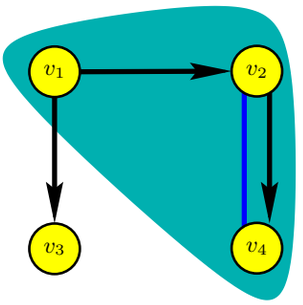
3.2 Ports

A *port* is a subset of the incidence relations of a node and can be viewed as a part of a node to which edges may attach. In electrical circuits for instance ports can be legs of a chip, and in graph drawing they may be used to specify points at which edges connect to a node. In GraphML `<port>`s appear as nested subelements of `<node>`s, and `<edge>`s may specify `<port>`s they attach to for both of their endpoints. See Fig. 4 for an example.




```
<graph edgedefault="directed">
  <desc>Mixed multigraph with node ports</desc>

  <node id="v1">
    <port name="north"/> <port name="east"/>
  </node>
  <node id="v2"/>
  <node id="v3"/>
  <node id="v4"/>

  <edge source="v1" sourceport="east" target="v2"/>
  <edge source="v1" target="v3"/>
  <edge source="v2" target="v4"/>
  <edge source="v2" target="v4" directed="false"/>
</graph>
```

Fig. 4. Ports have local names at vertices, and can be referenced separately by edges




```
<graph edgedefault="directed">
  <desc>Mixed multigraph with a hyperedge</desc>

  <node id="v1"/>
  <node id="v2"/>
  <node id="v3"/>
  <node id="v4"/>

  <edge source="v1" target="v2"/>
  <edge source="v1" target="v3"/>
  <edge source="v2" target="v4"/>
  <edge source="v2" target="v4" directed="false"/>

  <hyperedge>
    <endpoint node="v1" type="out"/>
    <endpoint node="v2" type="in"/>
    <endpoint node="v4"/>
  </hyperedge>
</graph>
```

Fig. 5. A hyperedge incident to v_1, v_2, and v_4, where v_1 is a source and v_2 is a sink

3.3 Hypergraphs

A *hyperedge* is a subset of nodes, together with a classification of these nodes into inputs, outputs, or neither of the two. A GraphML tag `<hyperedge>` may therefore contain any number of `<endpoint>`s which, in turn, refer to `<node>`s, but also classify these nodes using the XML attribute `type`. See Fig. 5 for the addition of hyperedges to a mixed multigraph.

Hyperedges are generalizations of edges, and edges could hence be represented as hyperedges. We have chosen to separate the two concepts for the benefit of applications that do not support hyperedges. By using two different XML tags, parsers can easily distinguish the two cases and invoke special treatment of `<hyperedge>`s if needed. Such applications may choose, e.g., to represent hyperedges using dummy nodes or to ignore them altogether.

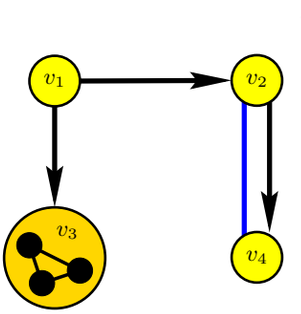
3.4 Nested Graphs

A *nested graph* is a graph occurring in an element of another graph. There are many models of hierarchical graphs, e.g., allowing more than one nested graph per element or a graph to be contained in more than one element.

Each item of a graph, i.e. each `<node>`, `<edge>`, or `<hyperedge>` may contain one nested `<graph>` element. Though simple, this model is sufficiently general to support all of the above variants. More than one contained graph can be expressed by defining a single contained graph which has a node for each of the child elements, and a contained graph appearing in different places can be referenced using a `<locator>` element.

For generality, we make no restrictions in the format as to which elements may be adjacent. GraphML supports edges between graphs, edges between elements of graphs at different levels of the containment hierarchy, etc., and leaves it to the application to detect inconsistencies with respect to its own model.

See Figure 6 for an example of a nested mixed multigraph.




```
<graph edgedefault="directed">
  <desc>Mixed multigraph with a nested graph</desc>

  <node id="v1"/>
  <node id="v2"/>
  <node id="v3">
    <graph id="G8local">
      <locator xlink:href="http://domain.tld/graphs.xml#G8"/>
    </graph>
  </node>
  <node id="v4"/>

  <edge source="v1" target="v2"/>
  <edge source="v1" target="v3"/>
  <edge source="v2" target="v4"/>
  <edge source="v2" target="v4" directed="false"/>
</graph>
```

Fig. 6. A nested graph (located in another document)

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE graphml SYSTEM "graphml.dtd">
<graphml>
  <graph edgedefault="directed">
    <desc>The entire example graph</desc>

    <node id="v1">
      <port name="north"/>
      <port name="east"/>
    </node>
    <node id="v2"/>
    <node id="v3">
      <graph id="G8">
        <locator xlink:href="http://domain.tld/graph.xml#G8"/>
      </graph>
    </node>
    <node id="v4"/>

    <edge source="v1" sourceport="east" target="v2"/>
    <edge source="v1" target="v3"/>
    <edge source="v2" target="v4"/>
    <edge source="v2" target="v4" directed="false"/>

    <hyperedge>
      <endpoint node="v1" port="north" type="out"/>
      <endpoint node="v2" type="in"/>
      <endpoint node="v4"/>
    </hyperedge>
  </graph>
</graphml>
```

Fig. 7. A complete GraphML document representing the graph of Fig. 2

In this section, we have described how mixed multigraphs, possibly with node ports, hyperedges and nested graphs, are represented in GraphML and thus completed the structural layer. The entire graph of Fig. 2 is stored in the GraphML document in Fig. 7. It remains to show how data not related to the structure of the graph is incorporated into a GraphML document.

4 Additional Data

The structural layer described in the previous section separates – both conceptually and in XML terms – the structure of a graph from every other type of data related to it. We propose the placement of additional data in well-defined locations without prescribing the representation of of the data. These locations are defined with the help of `<key>` and `<data>` tags. Furthermore, we propose means to structure and type the content of `<data>` tags.

4.1 Unstructured Data

Data labelings are considered to be (partial) functions that assign values in an arbitrary range to elements of the graph (which usually have the same type). Edge weights, for instance, can be viewed as a function from the set of edges into, say, the real numbers. For each such function, GraphML requires a `<key>`, providing a name and a domain (via the XML attribute `for`) for the class of labels. The optional content of a `<key>` tag is used as the function's default value.

Each element in the GraphML structural layer – except for `<locator>` – may contain any number of `<data>` tags, representing data values assigned to the corresponding graph item. A `<data>` tag refers to its `<key>`, i.e. the function, for which it provides a value, which in turn is defined by its content. If no `<data>` tag is present for an element in the domain of a given `<key>`, the default value is assumed.

Unless explicitly defined otherwise, the range of data labels is not restricted (i.e. `#PCDATA`).

4.2 Structured Data

More structured content can be defined by replacing the XML content model shared by `<key>` and `<data>` tags with a self-defined one. The mechanism proposed for such variations mimicks the W3C Recommendation for XHTML Modularization [5], which is currently implemented with DTDs only. The W3C is working on an implementation using XML Schema (see [9] for a primer), which we will adapt as soon as it becomes a Recommendation.

The DTD implementation is as follows. A parameter entity `\%GRAPHML.data.content`, initially defined to be `\#PCDATA`, is used to specify the content model of `<key>` and `<data>` tags. A GraphML extension module, say EXT, would define new XML tags to represent data specific to

the corresponding application and a parameter entity `\%EXT.data.content` specifying a content model for this module.

Recall that there may be more than one module. A driver file therefore defines the GraphML variant resulting from a specific combination of modules by importing them into GraphML and overwriting the content models of `<key>` and `<data>` with a combination of the content models defined in the modules. For details and examples see the GraphML homepage [2] and the XHTML Modularization Tutorial [6].

In effect, arbitrary variants of GraphML can be created by specifying valid XML for the content of `<key>` and `<data>` tags in extension modules, but the structural definition of the graph remains unaltered, regardless of the structure and type of data associated with it.

4.3 Typed Data

A second issue important for parsers and human interpretation alike is the typing of data labels. While the mechanism outlined in the previous subsection ensures that the XML content of `<data>` tags is any valid data item defined in an extension module, we suggest a way to infer the intended data type prior to having read all `<data>` tags that refer to the same `<key>`.

Each GraphML tag has an associated parameter entity in its list of XML attributes. This parameter is empty by default, but can be overwritten by extensions, thus adding new XML attributes to GraphML tags. In particular, one can define new XML attributes for `<key>`s. A GraphML extension module can therefore define an XML attribute with enumeration type values that correspond to meaningful types to content. It is likely that, when modularization via XML Schema becomes available, part of the type-checking can be delegated down to the XML level.

In summary, extensions can define any XML format to describe additional data and they can also provide typing information for parsers to deal with data in an efficient and type-safe manner. Multiple extensions can be combined freely, and extensions need not be supported by a parser to correctly extract contained graphs, since extensions are confined to subtrees of the Document Object Model (DOM) rooted at `<data>` tags and therefore cannot interfere with the structural layer.

5 Summary and Future Work

We have proposed GraphML, an XML format for the exchange of graph data. GraphML has been designed to be simple, general, robust, and, in particular, extensible. Application developers may define GraphML variants to include application-specific data, thereby making full use of XML's capabilities, but the design of GraphML ensures that such extensions are transparent to systems unaware of the extension. A tutorial on extending GraphML is in preparation.

Up-to-date information on GraphML is available on the Web [2], together with experimental parsers and graph editors using them. After a public review period, the specification will be finalized and published in full. To avoid even further diversification of formats, we aim towards the integration of GraphML with GXL (Graph Exchange Language) [3].

GraphML lays the ground towards an exchange format for graph visualizations. Future work will concentrate on defining GraphML extension modules for various applications, in particular graph drawing and information visualization. An important option considered in our preliminary designs for such modules is to make use of SVG (Scalable Vector Graphics) [4] in some form or another.

Acknowledgments. Many people have contributed to the current proposal. We would especially like to thank Stephen North and Roberto Tamassia for their continuing support and expertise, Jürgen Lerner and Sascha Meinert for implementing experimental parsers, John Punin for translating the GraphML DTD into XML Schema, Giuseppe Liotta for running the graph format panel discussion at GD 2001, and Petra Mutzel for including it into the program as well as giving us the opportunity to publish this progress report in the proceedings. We would also like to express our thanks to all members active or passive of the GraphML mailing list for their support of the project. In addition to all those mentioned before these are Vladimir Batagelj, Anne-Lise Gros, Carsten Gutwenger, David Jensen, Serban Jora, Michael Kaufmann, Guy Melançon, Maurizio Patrignani, Tim Pattison, Matthew Phillips, John Punin, Susan Sim, Adrian Vasiliu, Vance Waddle, and Andreas Winter.

References

1. Ulrik Brandes, M. Scott Marshall, and Stephen C. North. Graph data format workshop report. In Joe Marks, editor, *Proceedings of the 8th International Symposium on Graph Drawing (GD 2000)*, volume 1984 of *Lecture Notes in Computer Science*, pages 407–409. Springer, 2001.
2. GraphML homepage. `http://www.graphdrawing.org/graphml/`.
3. Ric Holt, Andy Schürr, Susan Sim, Andreas Winter. GXL – Graph eXchange Language. `http://www.gupro.de/GXL/`. Also refer to the article in this volume.
4. W3C. Scalable Vector Graphics (SVG). `http://www.w3.org/Graphics/SVG`.
5. W3C. XHTML^{TM} 1.1 – Module-based XHTML. `http://www.w3.org/TR/2001/REC-xhtml11-20010531/`.
6. W3C. XHTML Modules and Markup Languages. `http://www.w3.org/MarkUp/Guide/xhtml-m12n-tutorial/`.
7. W3C. Extensible Markup Language (XML). `http://www.w3.org/XML/`.
8. W3C. XML Protocol Activity. `http://www.w3.org/2000/xp/`.
9. W3C. XML Schema Part 0: Primer. `http://www.w3.org/TR/xmlschema-0/`.

A GraphML Document Type Definition

The following is a simplified version of the proposed Document Type Definition (DTD) for GraphML. The omitted details are relevant only for extension module developers and given on the GraphML homepage [2].

```
<!-- documents --------------------------------------------------------------
  GraphML documents start with an optional content description,
  followed by the declaration of any number of keys and a sequence of graphs.
  ------------------------------------------------------------------------->
<!ELEMENT graphml ((desc)?,(key)*,(graph)*)>


<!-- comments ---------------------------------------------------------------
  A description element contains human-readable text
  describing the content of the element it appears in.
  ------------------------------------------------------------------------->
<!ELEMENT desc (#PCDATA)>


<!-- remote definitions -----------------------------------------------------
  A locator may be used instead of other content of a graph or data element
  to refer to the location of the actual definition of the enclosing item's
  content.
  ------------------------------------------------------------------------->
<!ELEMENT locator EMPTY>
<!ATTLIST locator
  xmlns:xlink CDATA    #FIXED    "http://www.w3.org/TR/2000/PR-xlink-20001220/"
  xlink:href  CDATA    #REQUIRED
  xlink:type  (simple) #FIXED    "simple"
>


<!-- graphs -----------------------------------------------------------------
  A graph contains an optional description, keys local to this graph,
  and either a locator indicating that the graph is defined elsewhere,
  or lists of nodes, (hyper)edges, and data associated with the graph
  (in any order).
  A graph may be identified using the "id" attribute.  The mandatory attribute
  "edgedefault" indicates whether edges are directed or undirected by default;
  this can be overwritten locally by every edge.
  ------------------------------------------------------------------------->
<!ELEMENT graph ((desc)?,(key)*,((((data)|(node)|(edge)|(hyperedge))*)|(locator)))>
<!ATTLIST graph
  id          ID                    #IMPLIED
  edgedefault (directed|undirected) #REQUIRED
>


<!-- nodes ------------------------------------------------------------------
  Each node in a graph has to have a (unique) id.
  It may contain a description, followed by a sequence of ports and node
  data in any order, and may contain another graph.  Alternatively,
  it can be defined in another location, including a different file.
  Ports are identified by a name which does not have to be unique throughout
  the document, but within a node.  They can be nested hierarchically.
  ------------------------------------------------------------------------->
<!ELEMENT node ((desc)?,((((data)|(port))*,(graph)?)|(locator)))>
<!ATTLIST node id ID #REQUIRED>

<!ELEMENT port ((desc)?,((data)|(port))*)>
<!ATTLIST port name NMTOKEN #REQUIRED>



<!-- edges ------------------------------------------------------------------
  Similiar to nodes, edges may contain a separate description, followed by
  any number of edge data and, potentially, a nested graph.
  An edge must refer to a source and a target node, and may specify ports
  it attaches to.  However, such ports are not implicitly created and must
  therefore be defined at the corresponding node.
  Using the attribute "directed", the default value defined for the enclosing
  graph can be overwritten.
```

```
  ------------------------------------------------------------------------>
<!ELEMENT edge ((desc)?,(data)*,(graph)?)>
<!ATTLIST edge
  id         ID           #IMPLIED
  source     IDREF        #REQUIRED
  sourceport NMTOKEN      #IMPLIED
  target     IDREF        #REQUIRED
  targetport NMTOKEN      #IMPLIED
  directed   (true|false) #IMPLIED
>


<!-- hyperedges ------------------------------------------------------------
  Since the number of nodes incident to a hyperedge is arbitrary, they are
  not referred to via attributes of hyperedge.  Rather, a child element
  endpoint is created for each incident node, which refers to the node and,
  optionally, the port it is incident to.  For each incidence, it can be
  specified separately whether the node is a source ("out"), a target ("in"),
  or neither ("undir").
  ------------------------------------------------------------------------>
<!ELEMENT hyperedge ((desc)?,((data)|(endpoint))*,(graph)?)>
<!ATTLIST hyperedge id ID #IMPLIED>

<!ELEMENT endpoint ((desc)?)>
<!ATTLIST endpoint
  id   ID             #IMPLIED
  node IDREF          #REQUIRED
  port NMTOKEN        #IMPLIED
  type (in|out|undir) "undir"
>


<!-- additional data -------------------------------------------------------
  Additional data can be attached to any GraphML item by inserting data tags.
  To distinguish different sorts of data, those of the same sort refer
  to a common key tag.  A key may specify the domain it is valid for,
  and may contain a default value for that domain.  A key can thus be seen
  as the declaration of an array, non-default values of which are defined
  by the respective element.
  Extension modules may overwrite the common content model of key and data
  and add new attributes to keys to provide data type information.
  ------------------------------------------------------------------------>
<!ENTITY % GRAPHML.key.attrib "">
<!ENTITY % GRAPHML.data.content "(#PCDATA)">

<!ELEMENT key %GRAPHML.data.content;>
<!ATTLIST key
  id  ID                                          #REQUIRED
  for (graph|node|edge|hyperedge|port|endpoint|all) "all"
  %GRAPHML.key.attrib;
>

<!ELEMENT data %GRAPHML.data.content;>
<!ATTLIST data
  key IDREF #REQUIRED
  id  ID    #IMPLIED
>
```