GXL to GraphML and Vice Versa with XSLT [1]

Ulrik Brandes Jürgen Lerner Christian Pich [2]

Department of Computer & Information Science
University of Konstanz
78457 Konstanz, Germany

Abstract

We explore the issues involved in converting graph data stored in GXL or GraphML into each other. It turns out that XSLT provides a simple, portable, and effective mechanism for format conversion in either direction. As a by-product, some subtle differences between the formats become apparent.

Key words: Graph data formats, GraphML, GXL, XSLT

1 Introduction

Over the last years, development of generally applicable formats for representing graphs as files or streams has concentrated on only a few languages, the most recent of which are based on XML, thus taking advantage of a great deal of related standards and tools.

While the Graph Exchange Language (GXL) [4] largely originates in the field of software engineering, modeling techniques, and corresponding tools, the Graph Markup Language (GraphML) [1] has its background in the graph drawing community, and focuses on generality and extensibility.

Often, there is a need for converting graphs from one format to another with as little structural and syntactical mismatch as possible. We found Extensible Stylesheet Language Transformations (XSLT) [3] ideal for the specification of such transforms between XML languages, since the style sheets are portable, easily customizable, and open to graph format extensions. XSLT transforms an input XML document tree to an output XML document tree with a recursive pattern-matching mechanism. Such transforms are easy to add to existing applications and services as graph format translation filters, as described e.g. in [2].

[1] Research partially supported by DFG under grant Br 2158/1-2 and EU under grant IST-2001-33555 COSIN.

[2] Corresponding author: Christian.Pich@uni-konstanz.de

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

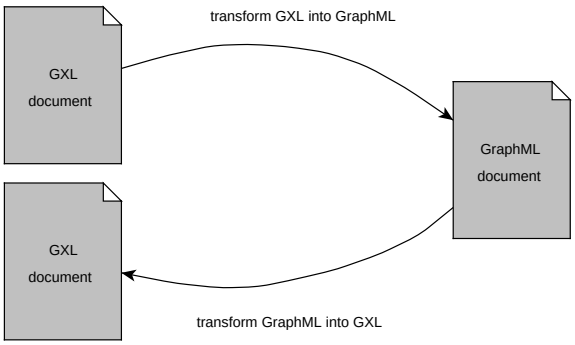

Fig. 1. Converting an original GXL file to GraphML and back. Note that the finally resulting GXL is not necessarily equivalent to the original GXL document because of the intermediate step via GraphML

We are particularly interested in a scenario in which data represented in GXL format is converted into a GraphML document that is used as input for a tool, e.g. a layout service, and finally converted back into a GXL document while preserving, for instance, custom data labels added by the tool, describing the layout. See Fig. 1.

This article is organized as follows. In Sect. 2, the two formats are compared in terms of expressivity and compatibility. In Sects. 3 and 4, we outline XSLT transformations from GXL to GraphML and back, respectively. Finally, Sect. 5 gives a brief discussion of relations and transformations between the two formats.

2 Comparison of GraphML and GXL

2.1 Basic Graph Models

Typically, graph description formats have their set of supported graph models centered around a structural core of basic graph classes common to most applications, describing a combinatorial structure of nodes (objects) connected by edges (relations).

In both GraphML and GXL, nodes are represented as an unordered list of `<node>` elements, each having an obligatory unique identifier, and edges between nodes as `<edge>` elements with references to the corresponding source and target identifiers. This is so straightforward that the syntax is almost the same in both formats, aside from some differences in attribute names.

Edges can be explicitly declared as directed or undirected, and directed and undirected edges may be freely mixed. In addition, both languages offer an attribute at `<graph>` level to set a default for all contained edges. Especially for treating multiedges (i.e. more than one edge between the same pair of nodes), there is the option to give identifiers to the edges to make them uniquely accessible.

GXL supports ordering of incidences with `fromorder` and `toorder` attributes, i.e. some or all of the edges connect to a node in a particular order, for which there is no directly corresponding element or attribute in GraphML. However, GraphML offers `<port>`s as specialization of incidence relations.

2.2 Advanced Graph Models

Sometimes conventional graphs are not rich enough to express particular models. Therefore, GraphML and GXL are aware of hyperedges, which can be seen as a generalization of conventional edges, and relate an arbitrary number of nodes, e.g. to model clusters or other distinguished sets of nodes. This concept is realized in GraphML by element `<hyperedge>` containing `<endpoint>`s and in GXL by `<rel>` and `<relend>`s. Both languages allow to mark the endpoints as a source or a sink in the respective hyperedge.

Nested graphs, i.e. graphs with nodes that contain other graphs, can be expressed in both languages, as well. This is especially useful when a complex graph integrates various abstraction levels. Syntactically, this is done by `<node>`s recursively containing another `<graph>`, where GraphML and GXL both offer an element `<locator>` to link to internal or external graphs.

2.3 Additional Data

Usually, applications need to enrich nodes and edges with further additional information, such as edge weights, layout attributes, or labels for names. In GXL, graph elements can be attributed with `<attr>` children containing additional data; their content can be a scalar type enclosed by `<string>`, `<int>`, `<float>`, or `<bool>`, or a built-in composite type, i.e. an (un-)ordered sequence of scalar or composite types, such as `<seq>` sequences or `<tup>` tuples. The obligatory `name` attribute serves to indicate `<attr>` elements of the same purpose.

Likewise, GraphML allows `<data>` labels to be associated with graph elements, together with unique names. However, it does not require the type to be contained as a markup element in each corresponding `<data>`; rather, the `key` attribute refers to a valid identifier of a `<key>` element that is in scope for the current `<graph>`. Thus, the type of that attribute and further information (e.g., whether the key is valid for edges, nodes, etc., or all graph elements) can be globally defined for the whole document at only one place.

Moreover, GraphML allows `<key>` definitions to include a `<default>` element that contains a default value for graph elements for which no corresponding `<data>` is present.

2.4 Other Concepts

Both GraphML and GXL offer means to distribute graphs and data over more than one document. While a GraphML `<locator>` serves to replace

GXL	Comment	GraphML
`<gxl>`	Substitutable. Both allow an arbitrary number of `<graph>`s per document.	`<graphml>`
`<graph>`	Substitutable. In GraphML, all `<data>` elements of the same kind are declared at this level using a `<key>`.	`<graph>`
`<node>`	Substitutable. May contain `<port>`s in GraphML.	`<node>`
`<edge>`	Substitutable. GXL incidences may be ordered, while GraphML edges may connect to `<port>`s defined in the corresponding `<node>`s.	`<edge>`
`<attr>`	In GXL, content must be enclosed by a tag locally indicating its type, (scalar such as `<string>`, or composite such as `<set>` or `<tup>`), while GraphML requires a reference to a global declaration using `<key>`.	`<data>`
–	Must be created when converting from GXL to GraphML.	`<key>`
–	Must be resolved when converting from GraphML to GXL.	`<default>`
`<rel>`	Substitutable. Both connect to `<node>`s via directed or undirected incidence.	`<hyperedge>`
`<relend>`	Substitutable. GraphML incidences are unordered; an ordering can be simulated using numbered `port`s.	`<endpoint>`
–	Partitioning of incidences; more general than the ordering of incidences in GXL.	`<port>`
`<type>`	Can be expressed with a designated `<data>` element in GraphML.	–
`<locator>`	Different. A reference to an arbitrary external data object in GXL, but a reference to a `<graph>` or `<node>` object stored in a different location (in the same or another document) in GraphML.	`<locator>`
–	Textual descriptions must be represented in GXL with `<attr>` labels or as XML comments `<!-- -->`.	`<desc>`

Fig. 2. Element-level comparison of GXL and GraphML

```
<xsl:template match="graph">
  <graph id="{@id}">
    <xsl:attribute name="edgedefault">
      <xsl:if test="contains(@edgemode,'un')">un</xsl:if>
      <xsl:text>directed</xsl:text>
    </xsl:attribute>
    <!-- generate a key (at the first appearance of an attr-name) -->
    <xsl:for-each select=".//attr[not(@name=../preceding-sibling::*/attr/@name)]">
      <key id="{@name}">
        <xsl:variable name="name" select="@name"/>
        <xsl:attribute name="for">  <!-- determine "minimal" domain -->
          <xsl:choose>
            <xsl:when test="not(@name=../../*[name()!='node']/attr/@name)">node</xsl:when>
            <xsl:when test="not(@name=../../*[name()!='edge']/attr/@name)">edge</xsl:when>
            <xsl:when test="not(@name=../../*[name()!='rel']/attr/@name)">hyperedge</xsl:when>
            <xsl:when test="not(@name=../../*[name()!='graph']/attr/@name)">graph</xsl:when>
            <xsl:when test="not(@name=../../*[name()!='relend']/attr/@name)">endpoint</xsl:when>
            <xsl:otherwise>all</xsl:otherwise>
          </xsl:choose>
        </xsl:attribute>
      </key>
    </xsl:for-each>
    <xsl:apply-templates select="*"/>
  </graph>
</xsl:template>
```

Fig. 3. GraphML to GXL: A template for a graph. A key corresponds to the first occurrence of an attribute name in the GXL source document; the domain of a key is determined by checking in which types of graph element it is used.

`<graph>`s or `<data>` in the same or another document, the GXL `<locator>` is another scalar data type whose content is an XLink reference to arbitrary URIs. Comments on the graph, i.e. textual descriptions for particular elements, are represented in GraphML as `<desc>` children, while in GXL they must be an `<attr>` child, or an XML comment enclosed by `<!-- -->`.

One of the most prominent design goals for GraphML and GXL is openness to application-specific extensions; their DTD or XML Schema definition may be redefined and extended by custom elements and attributes, which is especially interesting and useful when applications need to use complex XML subtrees as content instead of only scalar values.

Figure 2 summarizes the comparison of the core elements of both languages, together with a short comment on how they are interpreted and translated to an equivalent counterpart. The next two sections outline the main issues we had to face when converting from one graph format to the other with XSLT, trying to keep the structural mismatch minimal. Complete style sheets are given in Appendix A.

3 Transforming GXL into GraphML

Translating a GXL document to GraphML is best done with a pattern-oriented style sheet because the general structure of the two languages is very similar. Mapping the basic `<gxl>`, `<graph>`, `<node>`, and `<edge>` elements is done with respective templates that recursively initiate the pattern matching process

```
<xsl:attribute name="edgemode">
  <xsl:choose>
    <xsl:when test="@edgedefault='undirected'">
      <xsl:choose> <!-- are there directed edges overriding default? -->
        <xsl:when test="edge[@directed='true']">defaultundirected</xsl:when>
        <xsl:otherwise>undirected</xsl:otherwise>
      </xsl:choose>
    </xsl:when>
    <xsl:otherwise>
      <xsl:choose> <!-- are there undirected edges overriding default? -->
        <xsl:when test="edge[@directed='false']">defaultdirected</xsl:when>
        <xsl:otherwise>directed</xsl:otherwise>
      </xsl:choose>
    </xsl:otherwise>
  </xsl:choose>
</xsl:attribute>
```

Fig. 4. Determining the edge mode attribute of a GXL graph.

for child elements with `<xsl:appl-templates/>`. This mechanism is already powerful enough to cover nested graphs.

When transforming additional data from GXL's `<attr>` to GraphML's `<data>` element, only scalar content is taken into account, i.e. when the content is enclosed by a single `<string>`, `<int>`, etc., and does not include any of the GXL elements for structured content, such as `<seq>`, `<set>`, etc., or other custom XML elements for nested content. However, additional treatment of complex data requires only local template modifications.

Since GraphML data tags refer to keys, each occurring `name` attribute of an `<attr>` element must be represented as a `<key>` at graph level. The construction of the keys is shown in Fig. 3, which contains the template for mapping a GXL `<graph>` to its GraphML counterpart.

The ordering of incidences, which is indicated in GXL by the `fromorder` and `toorder` attributes of `<edges>`, is transferred to the GraphML result by using `<port>`s. If there are `<edge>`s that connect to a given `<node>` with an ordering attribute, a `<port>` having the ordinal number as its name (negative if the edge is outgoing) is added to that `<node>` in the GraphML output, together with corresponding port references at the output GraphML `<edge>`s.

Since `<rel>` elements as n-ary relations or simply sets of `<relend>` objects are the same as `<hyperedge>`s in GraphML with their `<endpoint>`s, translating them from one language to another is mainly copying the elements after renaming some of their names and attributes. For the sake of simplicity, our style sheet ignores the ordering of incidences in hyperedges, as well as `<locator>`s, `<type>`s, and attributes `role` and `kind`.

4 Transforming GraphML into GXL

Style sheets transforming GraphML documents (back) to the GXL format are typically pattern-oriented, as well; converting elements `<graphml>`, `<graph>`, `<node>`, and `<edge>` to their GXL counterparts is straightforward and requires only very simple templates.

```
<xsl:template match="node">
  <node id="{@id}">
    <xsl:call-template name="resolve-default"/>
    <xsl:apply-templates/>
  </node>
</xsl:template>

<xsl:template name="resolve-default">
  <xsl:variable name="this" select="."/>
  <xsl:for-each select="ancestor-or-self::*/key[@for=name($this) or @for='all']
                        [not(@id=$this/data[@key])]/default">
    <attr name="{../@id}">
      <string><xsl:apply-templates/></string>
    </attr>
  </xsl:for-each>
</xsl:template>
```

Fig. 5. Template inserting default values, and another one calling it.

The `edgemode` attribute of a GXL `<graph>` is slightly more expressive than GraphML's `edgedefault` because it can be used to state whether `<edge>`s are allowed to override the given default value, `default`(`un`)`directed`, or not, (`un`)`directed`. It is determined by checking in the GraphML `<graph>` if there are edges contradictory to the `edgedefault` value. See Fig. 4 for the code.

GraphML `<port>`s are converted to GXL `fromorder` and `toorder` attributes if they contain a positive or negative number (interpreted as ports for incoming or outgoing edges), and ignored otherwise. If it is essential in the particular use case to preserve ports and references, they could also be realized as special GXL `<attr>` elements within the connecting `<edge>`s, or as special `<node>`s that are adjacent only to the "owner" of the ports.

When converting the GraphML `<data>` elements to GXL `<attr>` elements, again some special issues have to be taken care of. Since the core elements of GraphML do not carry any meta-information on the type of the content within their `<data>` elements, we converted everything to a `<string>`. Furthermore, any occurrence of structured data is ignored in either direction of our transformations, which is also customizable when necessary.

If there are any `<default>` values present in `<key>`s, they must be resolved in such a way that whenever there is an element whithin the `for`-domain of that key that does not have a corresponding `<data>` element, it is attributed with the content of the `<default>` element. Since this is a situation that can reoccur with more than one graph element type, we call a reusable named template as shown in Fig. 5.

Since the meaning of `<locator>` elements differs in GXL and GraphML, they cannot be mapped directly, and our style sheet ignores them. Alternatively, they can be resolved by physically including the referenced contents, i.e. copying referenced GraphML `<node>` or `<edge>` into the output document.

Finally, the `<desc>` element in GraphML is transformed to an XML comment, since there is no designated element in GXL for representing textual descriptions or comments. Alternatively, comments may be represented as `<attr>` children with a special name.

5 Discussion

Transforming between GXL and GraphML is best and most efficiently done with a transformation driven by pattern matching. In both directions, transforming within the structural layer is simple and direct, whereas converting additional data requires some carefulness. Moreover, it depends on the particular use case and application area whether and how data labels from the input should be preserved in the output.

Compatibility and interchangeability can be improved by extending both formats with tailor-made extension modules, so that format-specific information like attribute types is preserved when a graph is to be transformed into the other format, and later retransformed back to the original format.

References

[1] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall. GraphML progress report: Structural layer proposal. *Proc. 9th Intl. Symp. Graph Drawing (GD '01)*, *Lecture Notes in Computer Scienec* 2265:501–512. Springer, 2002.

[2] S. Bridgeman. GraphEx: An improved graph translation service. *Proc. 11th Intl. Symp. Graph Drawing (GD '03)*, *Lecture Notes in Computer Sciene* 2912:307–313. Springer, 2004.

[3] W3C. *XSL Transformations*. `http://www.w3.org/TR/xslt/`.

[4] A. Winter. Exchanging Graphs with GXL. *Proc. 9th Intl. Symp. Graph Drawing (GD '01)*, *Lecture Notes in Computer Sciene* 2265:485–500. Springer, 2002.

A XSLT Examples

For the sake of simplicity, XML namespaces are not considered in the following style sheets. To make them work correctly with GraphML and GXL documents containing namespaces, it is necessary to either add the namespaces to the patterns in the style sheets, or remove the namespace nodes in the document through a preprocessing step, typically with another (simple) style sheet.

A.1 GXL to GraphML

The following style sheet transforms a GXL input document into a GraphML output document as described in Sect. 3.

```
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes" encoding="iso-8859-1"/>
  <xsl:strip-space elements="*"/>

  <xsl:template match="gxl">
    <xsl:comment>
      This GraphML document was generated from GXL by
      a GXL-to-GraphML conversion style sheet.
    </xsl:comment>
    <graphml><xsl:apply-templates/></graphml>
  </xsl:template>

  <xsl:template match="graph">
    <graph id="{@id}">
      <xsl:attribute name="edgedefault">
        <xsl:if test="contains(@edgemode,'un')">un</xsl:if>
        <xsl:text>directed</xsl:text>
      </xsl:attribute>
      <!-- generate a key (at the first appearance of an attr-name) -->
      <xsl:for-each select=".//attr[not(@name=../preceding-sibling::*/attr/@name)]">
        <key id="{@name}">
          <xsl:variable name="name" select="@name"/>
          <xsl:attribute name="for">  <!-- determine "minimal" domain -->
            <xsl:choose>
              <xsl:when test="not(@name=../../*[name()!='node']/attr/@name)">node</xsl:when>
              <xsl:when test="not(@name=../../*[name()!='edge']/attr/@name)">edge</xsl:when>
              <xsl:when test="not(@name=../../*[name()!='rel']/attr/@name)">hyperedge</xsl:when>
              <xsl:when test="not(@name=../../*[name()!='graph']/attr/@name)">graph</xsl:when>
              <xsl:when test="not(@name=../../*[name()!='relend']/attr/@name)">endpoint</xsl:when>
              <xsl:otherwise>all</xsl:otherwise>
            </xsl:choose>
          </xsl:attribute>
        </key>
      </xsl:for-each>
      <xsl:apply-templates select="*"/>
    </graph>
  </xsl:template>

  <xsl:template match="node">
    <xsl:variable name="id" select="@id"/>
    <node id="{$id}">
      <xsl:for-each select="../edge[@from=$id]/@toorder">
        <port name="{-1*number(.)}"/>
      </xsl:for-each>
      <xsl:for-each select="../edge[@to=$id]/@fromorder">
        <port name="{.}"/>
      </xsl:for-each>
      <xsl:apply-templates/>
```

```
    </node>
  </xsl:template>

  <xsl:template match="edge">
    <edge source="{@from}" target="{@to}">
      <xsl:copy-of select="@id"/>
      <xsl:if test="@isdirected">
        <xsl:attribute name="directed">
          <xsl:value-of select="@isdirected"/>
        </xsl:attribute>
      </xsl:if>
      <xsl:if test="@fromorder">
        <xsl:attribute name="sourceport"><xsl:value-of select="-1*number(.)"/></xsl:attribute>
      </xsl:if>
      <xsl:if test="@toorder">
        <xsl:attribute name="targetport"><xsl:value-of select="."/></xsl:attribute>
      </xsl:if>
      <xsl:apply-templates/>
    </edge>
  </xsl:template>

  <!-- process only scalar data -->
  <xsl:template match="attr">
    <xsl:if test="count(*)=1">
      <data key="{@name}">
        <xsl:copy-of select="@id"/>
        <xsl:value-of select="*[1]"/>
      </data>
    </xsl:if>
  </xsl:template>

  <xsl:template match="rel">
    <hyperedge>
      <xsl:copy-of select="@id"/>
      <xsl:apply-templates/>
    </hyperedge>
  </xsl:template>

  <xsl:template match="relend">
    <endpoint node="{@target}">
      <xsl:attribute name="type">
        <xsl:choose>
          <xsl:when test="@direction='in'">in</xsl:when>
          <xsl:when test="@direction='out'">out</xsl:when>
          <xsl:otherwise>undir</xsl:otherwise>
        </xsl:choose>
      </xsl:attribute>
      <xsl:apply-templates/>
    </endpoint>
  </xsl:template>

  <xsl:template match="locator"/>
  <xsl:template match="type"/>
  <xsl:template match="seq|tup|bag|set"/>
</xsl:stylesheet>
```

A.2 GraphML to GXL

This style sheet transforms a GraphML document (back) to GXL as described in Sect. 4).

```
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes" encoding="iso-8859-1"/>
  <xsl:strip-space elements="*"/>

  <xsl:template match="key|default|port"/>

  <xsl:template match="graphml">
    <xsl:comment>
      This GXL document was generated from GraphML by
      a GraphML-to-GXL conversion style sheet.
    </xsl:comment>
    <gxl><xsl:apply-templates/></gxl>
  </xsl:template>

  <xsl:template match="graph">
    <graph edgeids="{not(edge[not(@id)])}" hypergraph="{boolean(hyperedge)}">
      <xsl:attribute name="id">
        <xsl:choose> <!-- copy ID if present, otherwise create one -->
          <xsl:when test="@id"><xsl:value-of select="@id"/></xsl:when>
          <xsl:otherwise><xsl:value-of select="generate-id()"/></xsl:otherwise>
        </xsl:choose>
      </xsl:attribute>
      <xsl:attribute name="edgemode">
        <xsl:choose>
          <xsl:when test="@edgedefault='undirected'">
            <xsl:choose> <!-- are there directed edges overriding default? -->
              <xsl:when test="edge[@directed='true']">defaultundirected</xsl:when>
              <xsl:otherwise>undirected</xsl:otherwise>
            </xsl:choose>
          </xsl:when>
          <xsl:otherwise>
            <xsl:choose> <!-- are there undirected edges overriding default? -->
              <xsl:when test="edge[@directed='false']">defaultdirected</xsl:when>
              <xsl:otherwise>directed</xsl:otherwise>
            </xsl:choose>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:attribute>
      <xsl:apply-templates/>
    </graph>
  </xsl:template>

  <xsl:template match="node">
    <node id="{@id}">
      <xsl:call-template name="resolve-default"/>
      <xsl:apply-templates/>
    </node>
  </xsl:template>

  <xsl:template match="edge">
    <edge from="{@source}" to="{@target}">
      <xsl:copy-of select="@id"/>
      <xsl:if test="@directed">
        <xsl:attribute name="isdirected">
          <xsl:value-of select="@directed"/>
        </xsl:attribute>
      </xsl:if>
      <xsl:if test="@sourceport[string(number(.))!='NaN']">
        <xsl:attribute name="fromorder">
          <xsl:value-of select="-1*number(@sourceport)"/>
        </xsl:attribute>
      </xsl:if>
```

```
      <xsl:if test="@targetport[string(number(.))!='NaN']">
        <xsl:attribute name="toorder">
          <xsl:value-of select="@targetport"/>
        </xsl:attribute>
      </xsl:if>
      <xsl:call-template name="resolve-default"/>
      <xsl:apply-templates/>
    </edge>
  </xsl:template>

  <!-- data labels with scalar content -->
  <xsl:template match="data[count(node()) = count(text())]">
    <attr name="{@key}">
      <xsl:copy-of select="@id"/>
      <string>
        <xsl:apply-templates/>
      </string>
    </attr>
  </xsl:template>

  <!-- ignore data labels with complex content -->
  <xsl:template match="data[count(node()) != count(text())]"/>

  <xsl:template match="hyperedge">
    <rel isdirected="{boolean(endpoint[@type != 'undir'])}">
      <xsl:copy-of select="@id"/>
      <xsl:call-template name="resolve-default"/>
      <xsl:apply-templates/>
    </rel>
  </xsl:template>

  <xsl:template match="endpoint">
    <relend target="{@node}">
      <xsl:if test="@type">
        <xsl:attribute name="direction">
          <xsl:choose>
            <xsl:when test="@type='in'">in</xsl:when>
            <xsl:when test="@type='out'">out</xsl:when>
            <xsl:otherwise>none</xsl:otherwise>
          </xsl:choose>
        </xsl:attribute>
      </xsl:if>
      <xsl:call-template name="resolve-default"/>
      <xsl:apply-templates/>
    </relend>
  </xsl:template>

  <xsl:template match="desc">
    <xsl:comment>GraphML desc: <xsl:value-of select="."/></xsl:comment>
  </xsl:template>

  <!-- named template to insert default values when no data tag is present
       for a key having a default value -->
  <xsl:template name="resolve-default">
    <xsl:variable name="this" select="."/>
    <xsl:for-each select="ancestor-or-self::*/key[@for=name($this) or @for='all']
                          [not(@id=$this/data[@key])]/default">
      <attr name="{../@id}">
        <string><xsl:apply-templates/></string>
      </attr>
    </xsl:for-each>
  </xsl:template>

  <xsl:template match="locator"/>
</xsl:stylesheet>
```