18

Graph Markup Language (GraphML)

Ulrik Brandes
University of Konstanz

Markus Eiglsperger
Zühlke Engineering

Jürgen Lerner
University of Konstanz

Christian Pich
Swiss Re

18.1 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) [BMN01]. 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.

Thanks to its XML syntax, GraphML can be used in combination with other XML based formats. On the one hand, its own extension mechanism allows to attach `<data>` labels with complex content (possibly required to comply with other XML content models) to GraphML elements. Examples of such complex data labels are Scalable Vector Graphics [W3Ca] describing the appearance of the nodes and edges in a drawing. On the other hand, GraphML can be integrated into other applications, e.g., in SOAP messages [W3Cb].

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

0-8493-8597-0/01/$0.00+$1.50

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 [DBETT99, STT81]. 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.

18.1.1 Related Formats

Besides GraphML there is a multitude of file formats for serializing graphs. Among the simplest ones are direct ASCII-based codings of tables (matrices) or lists, such as tab-separated value files. Specific instances of these include UCINET's `*.dl` files [BEF99] and Pajek's `*.net` files [DMB05]. XML-based formats to represent graphs include GXL [Win02], and DyNetML [TRC03].

18.2 Basic Concepts

In this section, we describe how graphs and simple graph data are represented in GraphML. The graph model used in this section is a *labeled mixed multigraph*, i. e. a tuple

$$G = (V, E, \mathcal{D}),$$

where V is a set of *nodes*, E a *multi-set* containing *directed* and *undirected edges*, and $\mathcal{D}$ a set of *data labels* that are partial functions from $\{G\} \cup V \cup E$ into some specified range of values. The data labels can encode, e. g., properties of nodes and edges such as graphical variables or, if nodes correspond to social actors, demographic characteristics such as gender or age. Thus, our graph model includes graphs that can contain both directed and undirected edges, loops, and multi-edges. This graph model will be extended in Sect. 18.3, where advanced concepts for the graph topology, like nested graphs, hypergraphs, and ports, are introduced. As an example, consider the document fragment and the graph it describes in Fig. 18.1.

```
<graphml>
  <graph edgedefault="directed">
    <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>
</graphml>
```

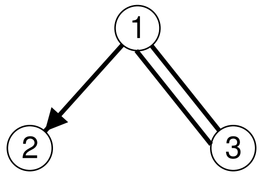

Figure 18.1 A graph and its representation in GraphML.

18.2.1 Header

The document fragment shown in Fig. 18.1 is not yet a *valid* XML document. Valid XML documents must declare in their header either a DTD (*document type definition*) or an XML schema. Both DTDs or schemas define a subset of all XML documents that forms a certain language. The GraphML language has been defined by a schema. Although a DTD is provided to support parsers that cannot handle schema definitions, the only normative specification is the GraphML schema located at

`http://graphml.graphdrawing.org/xmlns/1.1/graphml.xsd`

The document shown in Fig. 18.2 is minimal to be a GraphML document that can be validated against the above schema. Actually, it defines an empty set of graphs. Areas starting with `<!--` and ending with `-->` are comments.

```
<?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/1.1/graphml.xsd">
    <!--Content: List of graphs and data-->
</graphml>
```

Figure 18.2 A minimal valid GraphML document.

The first line of the GraphML document in Fig. 18.2 is an XML process instruction which 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.

The second line contains the *root-element*XS of a GraphML document: the `<graphml>` element. The `<graphml>` element, like all other GraphML elements, belongs to the namespace `http://graphml.graphdrawing.org/xmlns`. For this reason we define this namespace as the *default namespace* in the document by adding the XML Attribute

`xmlns="http://graphml.graphdrawing.org/xmlns"` to it. The next two XML Attributes declare which XML Schema is used for validation of this document. The attribute

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

defines `xsi` as the *namespace prefix* for the XML Schema namespace. The attribute,

```
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
                   http://graphml.graphdrawing.org/xmlns/1.1/graphml.xsd"
```

defines the *XML Schema location* for the GraphML namespace. It provides the information that all elements in the GraphML namespace are validated against the file `graphml.xsd` located at the given URL. Of course, validation is not necessarily performed using this file. Local copies of `graphml.xsd` can also be specified as schema locations. (Generally, the value of the `schemaLocation` attribute is a list of pairs, where the first element of each pair denotes a namespace and the second points to a file where elements of this namespace are defined.)

The XML Schema reference provides means to validate the document and is therefore strongly recommended. If validation is not considered necessary, the schema location declaration can be omitted. A minimal GraphML document without Schema reference is shown in Fig. 18.3. Note that this file is not a valid document according to the XML specification.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns" >
        <!--Content: List of graphs and data-->
</graphml>
```

Figure 18.3 A minimal GraphML document without a schema reference.

18.2.2 Topology

In this section, we describe how the basic graph-topology (nodes and edges) are represented in GraphML.

Remind the document fragment shown in Fig. 18.1. A graph is represented in GraphML by a `<graph>` element. The `<graphml>` element can contain any number of `<graph>`s. The nodes of a graph are represented by a list of `<node>` elements. Each node must have an `id` attribute. The edge set is represented by a list of `<edge>` elements. Edges and nodes may be ordered arbitrarily and it is not required that all nodes are listed before all edges. Clearly, the space requirement for storing a graph with n nodes and m edges in GraphML is in $\mathcal{O}(n + m)$.

Edges point to source- and target-nodes by the values of their attributes `source` and `target` respectively. It is ensured in the GraphML Schema specification that node-ids are unique within the enclosing `<graph>` and that the attribute values of the `source` and `target` attributes match the `id` of some `<node>` within the enclosing `<graph>`. The possibility of enforcing this constraint already in the definition of the GraphML language is one of the advantages of using XML schema instead of a DTD.

The `edgedefault` attribute of `<graph>` declares whether edges are understood as directed or undirected per default. Individual `<edge>`s can overwrite this default by setting the value of their `directed` attribute to `true` or `false` respectively.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml>
  <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>
    <edge id="e2" source="n1" target="n3">
      <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>
```

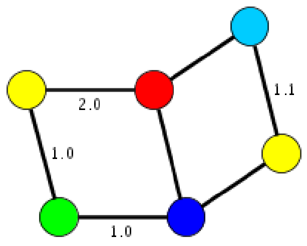

Figure 18.4 Graph with attributes. Edges have weights and nodes have colors. (For readability, the namespace declarations and schema location information has been left out.)

18.2.3 Attributes

In the previous section we discussed how to describe the topology of a graph in GraphML. While pure topological information may be sufficient for some applications of GraphML, for most of the time additional information is needed. With the help of the extension *GraphML-Attributes* one can specify additional information of simple type for the elements of the graph. Simple type means that the information is restricted to scalar values, e. g. numerical values and strings. The GraphML-Attributes extension is already included in the file

```
http://graphml.graphdrawing.org/xmlns/1.1/graphml.xsd
```

thus the header of the following example file may look like the one in Sect. 18.2.1. GraphML-Attributes must not be confused with XML-attributes which are a different concept (putting it in a simple way, GraphML-Attributes add information to graphs, sets of graphs, or parts of graphs and XML-attributes add information to XML elements).

In most cases, additional information can and should be attached to GraphML elements by usage of GraphML-Attributes as described in this section. This assures readability for other GraphML parsers. If a custom data-format is necessary, then the GraphML language can be extended to include arbitrary data in well-defined places. How extensions can be defined is described in Sect. 18.4.

GraphML-Attributes are considered to be partial functions that assign values to elements of the graph (which often but not necessarily have the same type). For example edges weights can be viewed as a function from the set of edges E to the real numbers.

$$\text{weight}\colon E \to \mathbf{R} \ .$$

As a different example, node colors can be represented by a function from the set of nodes V to strings over a certain alphabet Σ.

$$\text{color}\colon V \to \Sigma^* \ .$$

To add data functions to graph elements, the GraphML *key/data* mechanism has to be used. A `<key>` element, at the beginning of the document, *declares* a new data function; more precisely, the `<key>` element specifies the function's id, name, domain, and range of values. The values of the function are *defined* by `<data>` elements.

The declaration of all data functions right at the beginning of the document has the benefit that parsers can build up appropriate data structures at the beginning of the parsing process. Likewise, parsers can recognize if some required data is missing. The GraphML document shown in Fig. 18.4 is an example illustrating the key/data mechanism. The *weight* function is declared in the line

```
<key id="d1" for="edge" attr.name="weight" attr.type="double"/>
```

A `<key>` has an XML attribute called `for` that specifies the *domain* of the data function. The attribute `for` may assume values like `graph`, `node`, `edge`, `graphml` and names of other graph element types introduced later in Sect. 18.3. The XML attribute `for` may also assume the value `all` having the meaning that these data labels can be attached to all graph elements. The attribute `for` as well as a unique `id` are mandatory for `<key>` elements. The GraphML-Attributes extension provides two more attributes for `<key>`: the attribute `attr.name`, which defines the name of the data function and is used by parsers to recognize "their" data, and the attribute `attr.type`, which specifies the range of the data values.

Possible values for `attr.type` are `boolean`, `int`, `long`, `float`, `double`, and `string` having the obvious meaning.

A parser that handles edge weights will typically, after parsing the above line, initialize some internal data structure that stores doubles for each edge. Conversely, a parser that does not know or does not need a function for edges with the name "weight" will simply ignore the associated `<data>` elements.

Values for the data functions are defined in `<data>` elements. For example, the code fragment

```
<edge id="e0" source="n0" target="n2">
  <data key="d1">1.0</data>
</edge>
```

defines a value of 1.0 as weight for the enclosing `<edge>`. The `<data>` elements point to `<key>`s by their `key` attribute. It is ensured in the GraphML schema that the value of the `key` attribute must match the `id` of some `<key>` element within the same document.

Since in general data labels are only partial functions, `<data>` elements need not be present for all edges. For example the edge

```
<edge id="e3" source="n3" target="n2"/>
```

does not define a value for the weight function. However, `<key>`s can define default values for the associated data function. For example

```
<key id="d0" for="node" attr.name="color" attr.type="string">
  <default>yellow</default>
</key>
```

declares a function named `color` on the set of nodes and defines `yellow` as the default node color. Thus, the node

```
<node id="n4"/>
```

is understood as being colored yellow. Nodes can overwrite the default by their `<data>` element. For instance, the node

```
<node id="n0">
  <data key="d0">green</data>
</node>
```

is colored green. The default mechanism serves to save space if many elements assume the same value.

18.2.4 Parseinfo

There is one more extension, called *GraphML-Parseinfo*, to the core structural part of GraphML. GraphML-Parseinfo makes it possible to write simple parsers that rely on additional information in the GraphML files. The GraphML-Parseinfo extension is already included in the file

`http://graphml.graphdrawing.org/xmlns/1.1/graphml.xsd`

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml>
  <graph id="G" edgedefault="directed"
         parse.nodes="11" parse.edges="12"
         parse.maxindegree="2"
         parse.maxoutdegree="3"
         parse.nodeids="canonical"
         parse.edgeids="free"
         parse.order="nodesfirst">
    <node id="n0" parse.indegree="0" parse.outdegree="1"/>
    <node id="n1" parse.indegree="0" parse.outdegree="1"/>
    <node id="n2" parse.indegree="2" parse.outdegree="1"/>
    <node id="n3" parse.indegree="1" parse.outdegree="2"/>
    <node id="n4" parse.indegree="1" parse.outdegree="1"/>
    <node id="n5" parse.indegree="2" parse.outdegree="1"/>
    <node id="n6" parse.indegree="1" parse.outdegree="2"/>
    <node id="n7" parse.indegree="2" parse.outdegree="0"/>
    <node id="n8" parse.indegree="1" parse.outdegree="3"/>
    <node id="n9" parse.indegree="1" parse.outdegree="0"/>
    <node id="n10" parse.indegree="1" parse.outdegree="0"/>
    <edge id="edge0001" source="n0" target="n2"/>
    <edge id="edge0002" source="n1" target="n2"/>
    <edge id="edge0003" source="n2" target="n3"/>
    <edge id="edge0004" source="n3" target="n5"/>
    <edge id="edge0005" source="n3" target="n4"/>
    <edge id="edge0006" source="n4" target="n6"/>
    <edge id="edge0007" source="n6" target="n5"/>
    <edge id="edge0008" source="n5" target="n7"/>
    <edge id="edge0009" source="n6" target="n8"/>
    <edge id="edge0010" source="n8" target="n7"/>
    <edge id="edge0011" source="n8" target="n9"/>
    <edge id="edge0012" source="n8" target="n10"/>
  </graph>
</graphml>
```

Figure 18.5 Example demonstrating the use of *GraphML-Parseinfo* meta data.

thus the header of the example file in Fig. 18.5 may look like the one in Sect. 18.2.1.

To make it possible to implement optimized parsers for GraphML documents, meta-data can be attached as XML-attributes to some GraphML elements. There are two kinds of meta-data intended for parsers: information about the number of elements and information about how specific data is encoded in the document. For instance, a parser that stores nodes and incident edges in (non-extensible) arrays can profit from information about the number of nodes in the graph and the nodes' degrees, respectively. All XML-attributes denoting meta-data for parsers are prefixed with `parse`.

For the first kind, information about the number of elements, the following XML-attributes for the `<graph>` element are defined. The value of the attribute `parse.nodes` gives the number of `<node>`s in the `<graph>`. Likewise, the value of `parse.edges` gives the number of `<edge>`s, `parse.maxindegree` is for the maximum indegree of the all `<node>`s in the `<graph>`, and `parse.maxoutdegree` for the maximum outdegree. For `<node>` elements the value of the attribute `parse.indegree` gives the indegree and `parse.outdegree` the outdegree of `<node>`s, respectively.

For the second kind, information about element encoding, the following XML-attributes for the `<graph>` element are defined. If the attribute `parse.nodeids` has the value `canonical`, all `<node>`s have identifiers following the pattern `nX`, where `X` denotes the number of occurrences of `<node>` elements before the current element. Otherwise the value of `parse.nodeids` equals `free`. The same holds for `<edge>`s for which the corresponding XML-attribute `parse.edgeids` is defined, with the only difference that the identifiers of `<edge>`s follow the pattern `eX`. The XML-attribute `parse.order` of `<graph>` gives information about the order in which `<node>` and `<edge>` elements occur in the `<graph>`. If `parse.order` assumes the value `nodesfirst`, all `<node>` elements appear the first occurrence of an `<edge>`. If `parse.order` assumes the value `adjacencylist`, the declaration of a `<node>` is followed by the declaration of its adjacent `<edge>`s. If `parse.order` assumes the value `free`, no order is imposed. The example in Fig. 18.5 demonstrates the use of parse info meta-data.

18.3 Advanced Concepts

In this section we discuss advanced topological features for graphs. The graph model from Sect. 18.2 is extended to include a *nesting hierarchy*, *hyperedges* and *ports*. Since many graph applications do not support these extended graph models, we describe at the end of each subsection the specified fall-back behavior.

The GraphML elements that are introduced in this section can be specified as the domain of data-functions, i. e., as the value of the `for` attributes of `<key>`s (compare Sect. 18.2.3).

18.3.1 Nested Graphs

GraphML supports nested graphs, i.e., graphs in which the nodes are hierarchically ordered. The hierarchy tree is encoded in the GraphML document tree. A `<node>` in a GraphML document may contain a `<graph>` element which itself contains the `<node>`s which are in the hierarchy below this `<node>`. Figure 18.6 is an example of a document describing a nested graph. Note that in the drawing of the graph the hierarchy is expressed by containment, i. e., a node u is below a node v in the hierarchy if and only if the graphical representation of u is entirely inside the graphical representation of v.

The edges between two nodes in a nested graph have to be declared in a graph that is an ancestor of both nodes in the hierarchy. Note that this is true for our example. Declaring the edge between node `n3` and node `n2` inside graph `G1` would be wrong while declaring it

```
<graphml>
  <graph id="G0" edgedefault="undirected">
    <node id="n1">
      <graph id="G1" edgedefault="undirected">
        <node id="n3"/>
        <node id="n4"/>
        <node id="n5"/>
        <edge source="n3" target="n4"/>
        <edge source="n4" target="n5"/>
     </graph>
    </node>
    <node id="n2">
      <graph id="G2" edgedefault="undirected">
        <node id="n6"/>
     </graph>
    </node>
    <edge source="n1" target="n2"/>
    <edge source="n3" target="n2"/>
    <edge source="n3" target="n6"/>
  </graph>
</graphml>
```

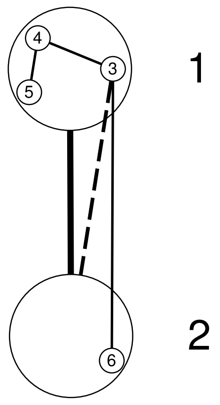

Figure 18.6 A nested graph.

in graph `G0` is correct. A good policy is to place the edges at the least common ancestor of the nodes in the hierarchy.

The GraphML language includes an element called `<locator>` which makes it possible to define some of the document content in another file. More specifically, the elements `<graph>` and `<node>` can contain a `<locator>` element whose attribute `xlink:href` points to a file in which the content of this `<graph>`, respectively `<node>` is defined. If a particular `<graph>` or `<node>` element contains a `<locator>`, then this `<graph>`, respectively `<node>` does not contain any other element. For instance, the document fragment

```
  <graph id="G0" edgedefault="undirected">
    <node id="n1">
      <graph id="G1" edgedefault="undirected">
        <locator xlink:href="content_of_G1.graphml"/>
     </graph>
    </node>
    ...
  </graph>
```

(which is a modified version of the document in Fig. 18.6) tells the parser that the content of the `<graph>` with `id="G1"` is defined in the file `content_of_G1.graphml`. Likewise, the content of `<node>`s can be outsourced to another file with the help of `<locator>` elements.

For applications that cannot handle nested graphs, the fall-back behavior 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.

18.3.2 Hypergraphs

Hyperedges are a generalization of edges in the sense that they do not only relate two endpoints to each other but rather 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 `id` of a `<node>` in the document. The example in Fig. 18.7 contains two hyperedges and two edges. The hyperedges are illustrated by joining arcs, the edges by straight lines.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml>
  <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>
```

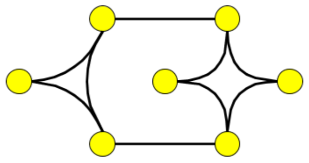

Figure 18.7 A hypergraph.

Note that edges can be either specified by an `<edge>` element or by a `<hyperedge>` element containing exactly two `<endpoint>` elements. Obviously, the latter option is only recommendable for applications that can handle hyperedges. The `<endpoint>` elements have an optional attribute called `type` which may assume the values `in`, `out`, and `undir` and is set to `undir` by default. The fall-back behavior for applications that can not handle hyperedges is simply to ignore them.

18.3.3 Ports

A node may specify different logical locations for edges and hyperedges to connect. The logical locations are called *ports*. As an analogy, think of the graph as a mother board, the nodes as integrated circuits and the edges as connecting wires. Then the pins on the integrated circuits correspond to ports of a node.

The ports of a node are declared by `<port>` elements as children of the corresponding `<node>` element. `<port>` elements may be nested, i. e., they may contain `<port>` elements themselves. Each `<port>` element must have an XML-attribute `name`, which is an identifier for this port. Port names are unique only within the enclosing `<node>` (see the example in Fig. 18.8). The `<edge>` element has optional XML-attributes `sourceport` and `targetport` with which an edge may specify the port on the source, resp. target, node. Correspondingly, the `<endpoint>` element has an optional XML-attribute `port`. An example of a GraphML document with ports is shown in Fig. 18.8. The fall-back behavior for applications that can not handle ports is simply to ignore them.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml>
  <graph id="G" edgedefault="directed">
    <node id="n0">
      <port name="North"/>
      <port name="South"/>
      <port name="East"/>
      <port name="West"/>
    </node>
    <node id="n1">
      <port name="North"/>
      <port name="South"/>
      <port name="East"/>
      <port name="West"/>
    </node>
    <node id="n2">
      <port name="NorthWest"/>
      <port name="SouthEast"/>
    </node>
    <node id="n3">
      <port name="NorthEast"/>
      <port name="SouthWest"/>
    </node>
    <edge source="n0" target="n3"
          sourceport="North" targetport="NorthEast"/>
    <hyperedge>
       <endpoint node="n0" port="North"/>
       <endpoint node="n1" port="East"/>
       <endpoint node="n2" port="SouthEast"/>
     </hyperedge>
  </graph>
</graphml>
```

Figure 18.8 Document of a graph with ports.

18.4 Extending GraphML

GraphML is designed to be easily extensible. With GraphML the topology of a graph and simple attributes of graph elements (see Sect. 18.2.3) can be serialized. To store more complex application data one has to extend GraphML which will be discussed in this section.

GraphML can be extended in two different ways: adding additional attributes to GraphML elements (discussed in Sect. 18.4.1) and extending the content of the `<data>` elements by allowing them to contain elements from other XML languages (discussed in Sect. 18.4.2).

Extensions of GraphML should be defined by an XML Schema (the other possibility, extending the DTD, is not described here). The Schema which defines the extension can be derived from the GraphML Schema documents by using a standard mechanism similar to the one used by XHTML.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
   targetNamespace="http://graphml.graphdrawing.org/xmlns"
   xmlns="http://graphml.graphdrawing.org/xmlns"
   xmlns:xlink="http://www.w3.org/1999/xlink"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified">

<xs:import namespace="http://www.w3.org/1999/xlink"
           schemaLocation="xlink.xsd"/>

<xs:redefine
  schemaLocation="http://graphml.graphdrawing.org/xmlns/1.1/graphml.xsd">
  <xs:attributeGroup name="node.extra.attrib">
    <xs:attributeGroup ref="node.extra.attrib"/>
    <xs:attribute ref="xlink:href" use="optional"/>
  </xs:attributeGroup>
</xs:redefine>

</xs:schema>
```

Figure 18.9 File `graphml+xlink.xsd` : an XML Schema Definition that extends the GraphML language by adding attribute `xlink:href` to element `<node>`.

18.4.1 Adding XML-attributes

In most cases, additional information can and should be attached to GraphML elements by usage of GraphML-Attributes (see Sect. 18.2.3). This assures readability for other GraphML parsers. However, sometimes it might be more convenient to use specific XML attributes. Suppose a graph whose nodes model WWW pages should be stored in GraphML. A node could then point to the associated page by storing the URL in an `xlink:href` attribute within the `<node>` element:

```
<node id="n0" xlink:href="http://graphml.graphdrawing.org"/>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
            xmlns:xlink="http://www.w3.org/1999/xlink"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
                                graphml+xlink.xsd">
  <graph edgedefault="directed">
    <node id="n0" xlink:href="http://graphml.graphdrawing.org"/>
    <node id="n1" />
    <edge source="n0" target="n1"/>
  </graph>
</graphml>
```

Figure 18.10 A document that can be validated with the XSD shown in Fig. 18.9. Note that the `schemaLocation` attribute of `<graphml>` points to the file `graphml+xlink.xsd`.

The string `http://graphml.graphdrawing.org` could as well be stored within a `<data>` element contained in the node `n0`. However, when storing this string as the value of the `xlink:href` attribute, then its semantic (being a URL) becomes more obvious.

The element `<node>` as written above would not be valid for the core GraphML, since there is no `xlink:href` attribute defined for `<node>`. To add XML attributes to GraphML elements one has to extend GraphML. This extension can be defined by an XML Schema. The document in Fig. 18.9 is an *XML Schema Definition* that extends the GraphML language by adding the `xlink:href` attribute to `<node>`.

The document in Fig. 18.9 has a `<schema>` element as its root element (every XML Schema Definition does so). The element `<schema>` has a couple of attributes:

```
targetNamespace="http://graphml.graphdrawing.org/xmlns"
```

specifies that the language defined by this document is GraphML. The next three lines specify the default namespace (identified by the GraphML URL) and the namespace prefixes for XLink and XMLSchema. The attributes `elementFormDefault` and `attributeFormDefault` are of no importance for this example.

The import instruction

```
<xs:import namespace="http://www.w3.org/1999/xlink"
           schemaLocation="xlink.xsd"/>
```

gives access to the XLink namespace (assumed that the Schema Definition for XLink is located at the file `xlink.xsd`).

The extension is done in the `<redefine>` element. The attribute

```
    schemaLocation="http://graphml.graphdrawing.org/xmlns/1.1/graphml.xsd"
```

of `<redefine>` specifies the file (part of) which is being redefined. The document fragment

```
  <xs:attributeGroup name="node.extra.attrib">
    <xs:attributeGroup ref="node.extra.attrib"/>
    <xs:attribute ref="xlink:href" use="optional"/>
  </xs:attributeGroup>
```

extends the *attribute group* called `node.extra.attrib` which (by the core GraphML specification) is an empty set, but included in the attribute-list of the element `<node>`. After redefinition, this attribute group has its old content plus one more attribute, namely `xlink:href`. This attribute is declared as being optional for `<node>`. It is a good policy to always add the old content to the newly defined attribute groups, as there might be more than one Schema definitions extending the same attribute group.

As there is the attribute group `node.extra.attrib` for the element `<node>`, there are corresponding attribute groups for all GraphML elements. These attribute groups are empty in the core GraphML definition but can be extended as illustrated above.

The schema `graphml+xlink.xsd` can be used to validate the document shown in Fig. 18.10.

Storing additional information directly in the attributes of GraphML elements, as illustrated in this section, may seem to be preferable to storing them within a `<data>` element, as explained in Sect. 18.2.3 (at least it can be observed that less characters are necessary). However, such a user-specified extension comes at a price: since these non-standard attributes are not declared by `<key>` elements, GraphML parsers might not be able to handle them.

18.4.2 Adding Structured Content

In some cases it might be convenient to use other XML languages to represent data in GraphML. For example a user wants to store images for nodes, written in SVG, as in the following document fragment.

```
   ...
xmlns:svg="http://www.w3.org/2000/svg"
   ...
<node id="n0" >
  <data key="k0">
    <svg:svg width="4cm" height="8cm" version="1.1">
      <svg:ellipse cx="2cm" cy="4cm" rx="2cm" ry="1cm" />
    </svg:svg>
  </data>
</node>
   ...
```

The attributes of `<svg>` and `<ellipse>` could also be stored in data functions as described in Sect. 18.2.3. However, the representation above is much more convenient, since applications can use existing parsers or viewers for SVG images.

GraphML can be extended to validate such a document. Arbitrary elements can be added to the content of `<data>`—but only to `<data>`—while the core GraphML cannot be changed. This decision has been made to ensure that parsers can understand at least the structural part and ignore possibly unknown content of `<data>`.

Figure 18.11 shows the XML Schema Definition that adds SVG elements to the content of `<data>`.

The schema in Fig. 18.11 is similar to the example in Fig. 18.9. First the namespace declarations are made. Then the SVG namespace is imported. As before, the extension is done in the `<redefine>` element. Within this element the *complex type* `data-extension.type` is extended by the SVG element `<svg>`. `data-extension.type` is the *base-type* for the content of the elements `<data>` and `<default>`. This type has empty content in the core GraphML definition, but can be extended by arbitrary XML elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
   targetNamespace="http://graphml.graphdrawing.org/xmlns"
   xmlns="http://graphml.graphdrawing.org/xmlns"
   xmlns:svg="http://www.w3.org/2000/svg"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified"
>

<xs:import namespace="http://www.w3.org/2000/svg"
           schemaLocation="svg.xsd"/>

<xs:redefine
    schemaLocation="http://graphml.graphdrawing.org/xmlns/1.1/graphml.xsd">
  <xs:complexType name="data-extension.type">
    <xs:complexContent>
      <xs:extension base="data-extension.type">
        <xs:sequence>
          <xs:element ref="svg:svg"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:redefine>

</xs:schema>
```

Figure 18.11 File `graphml+svg.xsd` : an XML Schema Definition that extends the GraphML language by adding element `<svg:svg>` to the content of `<data>`.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
            xmlns:svg="http://www.w3.org/2000/svg"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
                                graphml+svg.xsd">
  <key id="k0" for="node">
    <default>
      <svg:svg width="5cm" height="4cm" version="1.1">
        <svg:desc>Default graphical representation for nodes
        </svg:desc>
        <svg:rect x="0.5cm" y="0.5cm" width="2cm" height="1cm"/>
      </svg:svg>
    </default>
  </key>
  <key id="k1" for="edge">
    <desc>Graphical representation for edges
    </desc>
  </key>
  <graph edgedefault="directed">
    <node id="n0">
      <data key="k0">
        <svg:svg width="4cm" height="8cm" version="1.1">
          <svg:ellipse cx="2cm" cy="4cm" rx="2cm" ry="1cm" />
        </svg:svg>
      </data>
    </node>
    <node id="n1" />
    <edge source="n0" target="n1">
      <data key="k1">
        <svg:svg width="12cm" height="4cm" viewBox="0 0 1200 400">
          <svg:line x1="100" y1="300" x2="300" y2="100"
           stroke-width="5"  />
        </svg:svg>
      </data>
    </edge>
  </graph>
</graphml>
```

Figure 18.12 A document that can be validated with the XSD shown in Fig. 18.11. Note that the `schemaLocation` attribute of `<graphml>` points to `graphml+svg.xsd`.

Documents that are validated against the Schema in Fig. 18.11 can thus have `<data>` elements that contain `<svg>`. An example is shown in Fig. 18.12. The node with id `n1` admits the default graphical representation given within key `k0`. The above example shows also the usefulness of XML Namespaces. There are two different `<desc>` elements, one in the GraphML namespace and one in the SVG namespace. Possible conflicts, due to elements from different XML languages that happen to have identical names, are resolved by different namespaces.

We note that it is not only possible to use other XML languages (like SVG) within GraphML. GraphML can also be used to represent graph data within extensible XML languages like SVG or XHTML. The possibility to combine modularly built XML languages ensures the re-usability of parsers and other software. For example, SVG viewers could call graphdrawing software to layout graphs that are stored in GraphML within an SVG file.

18.5 Transforming GraphML

It is straightforward to provide access to graphs represented in GraphML by adding input and output filters to an existing software application. However, we find that Extensible Stylesheet Language Transformations (XSLT) [W3Cc] offer a more natural way of exploiting XML data, in particular when the resulting format of a computation is again based on XML. The mappings that transform input GraphML documents to output documents are defined in XSLT style sheets and can be used stand-alone, as components of larger systems, or in, say, web services [BP04].

Basically, the transformations are defined in style sheets (sometimes also called transformation sheets), which specify how an input XML document gets transformed into an output XML document in a recursive pattern matching process. The underlying data model for XML documents is the Document Object Model (DOM), a tree of DOM nodes representing the elements, attributes, text etc., which is held completely in memory. Fig. 18.13 shows the basic workflow of a transformation.

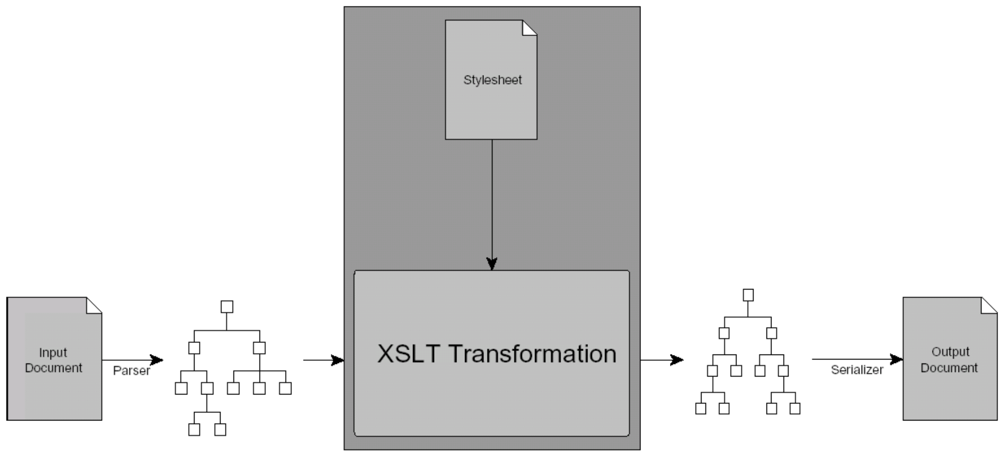

Figure 18.13 Workflow of an XSLT transformation. First, XML data is converted to a tree representation, which is then used to build the result tree as specified in the style sheet. Eventually, the result tree is serialized as XML. Taken from [BP04].

DOM trees can be navigated with the XPath language, a sublanguage of XSLT: It expresses paths in the document tree seen from a particular context node (similar to a directory tree of a file system) and serves to address sets of its nodes that satisfy given conditions. For example, if the context node is a `<graph>` element, all node identifiers can be addressed by `child::node/attribute::id`, or `node/@id` as shorthand. Predicates can be used to specify more precisely which parts of the DOM tree to select; for example, the XPath expression `edge[@source='n0']/data` selects only those `<data>` children of `<edge>`s starting from the `<node>` with the given identifier.

The transformation process can be roughly described as follows: A style sheet consists of a list of templates, each having an associated pattern and a template body containing the actions to be executed and the content to be written to the output. Beginning with the root, the processor performs a depth-first traversal (in document order) through the DOM tree. For each DOM node it encounters, it checks whether there is a template whose pattern it satisfies; if so, it selects one of the templates and executes the actions given in that template body (potentially with further recursive pattern matching for the subtrees), and does not do any further depth-first traversal for the DOM subtree rooted at that DOM node; else, it automatically continues the depth-first traversal recursively at each of its children. See Fig. 18.14 for an example of an XSLT transformation sheet.

18.5.1 Means of Transformation

The expressivity and usefulness of XSLT transformations goes beyond their original purpose of adding some style to the input. The following is an overview of some important basic concepts of XSLT and how these concepts can particularly be employed in order to formulate advanced GraphML transformations that also take into account the underlying combinatorial structure of the graph instead of only the DOM tree.

18.5.2 Transformation Types

Since GraphML is designed as a general format not bound to a particular area of application, an abundance of XSLT use cases exist. However, we found that transformations can be filed into three major categories, depending on the actual purpose of transformation. Note that there may of course be transformations that belong to more than one of these categories.

Internal While one of GraphML's design goals is to require a well-defined interpretation for all GraphML files, there is no uniqueness the other way round, i.e., there are various GraphML representations for a graph; for example, its `<node>`s and `<edge>`s may appear in arbitrary order. However, applications may require their GraphML input to satisfy certain preconditions, such as the appearance of all `<node>`s before any `<edge>` in order to set up a graph in memory on-the-fly while reading the input stream.

Generally, some frequently arising transformations include

- pre- and postprocessing the GraphML file to make it satisfy given conditions, such as rearranging the markup elements or generating unique identifiers,
- inserting default values where there is no explicit entry, e.g., edge directions or default values for `<data>` tags,
- resolving XLink references in distributed graphs,
- filtering out unneeded `<data>` tags that are not relevant for further processing and can be dropped to reduce communication or memory cost, and
- converting between graph classes, for example eliminating hyperedges, expanding

nested graphs, or removing multiedges.

```
<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:template match="data|desc|key|default"/> <!-- empty template-->

  <xsl:template match="/graphml">
    <graphml>
      <xsl:copy-of select="key|desc|@*"/>
      <xsl:apply-templates match="graph"/> <!-- process graph(s) -->
    </graphml>
  </xsl:template>

  <xsl:template match="graph">  <!-- override template -->
    <graph>
      <xsl:copy-of select="key|desc|@*"/>
      <xsl:copy-of select="node"/> <!-- nodes first -->
      <xsl:copy-of select="edge"/> <!-- then edges -->
    </graph>
  </xsl:template>
</xsl:stylesheet>
```

Figure 18.14 Example of an XSLT transformation sheet removing the elements `<data>`, `<desc>`, `<key>`, and `<default>` from the document and reorders nodes and edges such that all `<node>` elements appear before any `<edge>` element.

Format Conversion Although in recent years, GraphML and similar formats like GXL [Win02] and GML [GML] have become increasingly used in various areas of interest, there are still many applications and services not (yet) capable of processing them. To be compatible, formats need to be translatable to each other, preserving as much information as possible.

In doing so, it is essential to take into account possible structural mismatch in terms of both the graph models and concepts that can be expressed by the involved formats, and their support for additional data. Of course, the closer the conceptual relatedness between source and target format is, the simpler the style sheets typically are.

While conversion will be necessary in various settings, two use cases appear to be of particular importance:

- *Conversion into another graph format:* We expect GraphML to be used in many applications to archive attributed graph data and in Web services to transmit aspects of a graph. While it is easy to output GraphML, style sheets can be used to convert GraphML into other graph formats [BLP04] and can thus be used in translation services like GraphEx [Bri04].
- *Export to some graphics format:* Of course, graph-based tools in general and graph drawing tools in particular will have to export graphs in graphics formats for visualization purposes.

The transformation need not be applied to a filed document, but can also be carried out in memory by applications that ought to be able to export in some target format. Note that, even though XSLT is typically used for mapping between XML documents, it can also be utilized to generate non-XML output.

Algorithmic Algorithmic style sheets appear in transformations which create fragments in the output document that do not directly correspond to fragments in the input document, i.e. when there is structure in the source document that is not explicit in the markup. This is typical for GraphML data: For example, it is not possible to determine whether or not a given `<graph>` contains cycles by just looking at the markup; some algorithm has to be applied to the represented graph.

To get a feel for the potential of algorithmic style sheets, we implemented some basic graph algorithms using XSLT, and with recursive templates, it proved powerful enough to formulate even more advanced algorithms. For example, a style sheet can be used to compute the distances from a single source to all other nodes or execute a layout algorithm, and then attach the results to `<node>`s in `<data>` labels.

18.5.3 Language Binding

We found that pure XSLT functionality is expressive enough to solve even more advanced GraphML related problems. However, it suffers from some general drawbacks:

- With growing problem complexity, the style sheets tend to become disproportionately verbose.
- Algorithms must be reformulated in terms of recursive templates, and there is no way to use existing implementations.
- Computations may perform poorly, especially for large input. This is often due to excessive DOM tree traversal and overhead generated by template instantiation internal to the XSLT processor.
- There is no direct way of accessing system services, such as date functions or data base connectivity.

Therefore, most XSLT processors allow the integration of extension functions implemented in XSLT or some other programming language. Usually, they support at least their native language. For example, Saxon [Sax] can access and use external Java classes since itself is written entirely in Java. In this case, extension functions are methods of Java classes available on the class path when the transformation is being executed, and get invoked within XPath expressions. Usually, they are static methods, thus staying compliant with XSLT's design idea of declarative style and freeness of side-effects. However, XSLT allows to create objects and to call their instance-level methods by binding the created objects to XPath variables.

The architecture shown in Fig. 18.15 consists of three layers:

- The style sheet that instantiates the wrapper and communicates with it
- A wrapper class (the actual XSLT extension) that converts GraphML markup to a wrapped graph object, and provides computation results
- Java classes for graph data structures and algorithms

Thus, the wrapper acts as a mediator between the graph object and the style sheet. The wrapper instantiates a graph object corresponding to the GraphML markup, and, for instance, applies a graph drawing algorithm to it. In turn, it provides the resulting coordi-

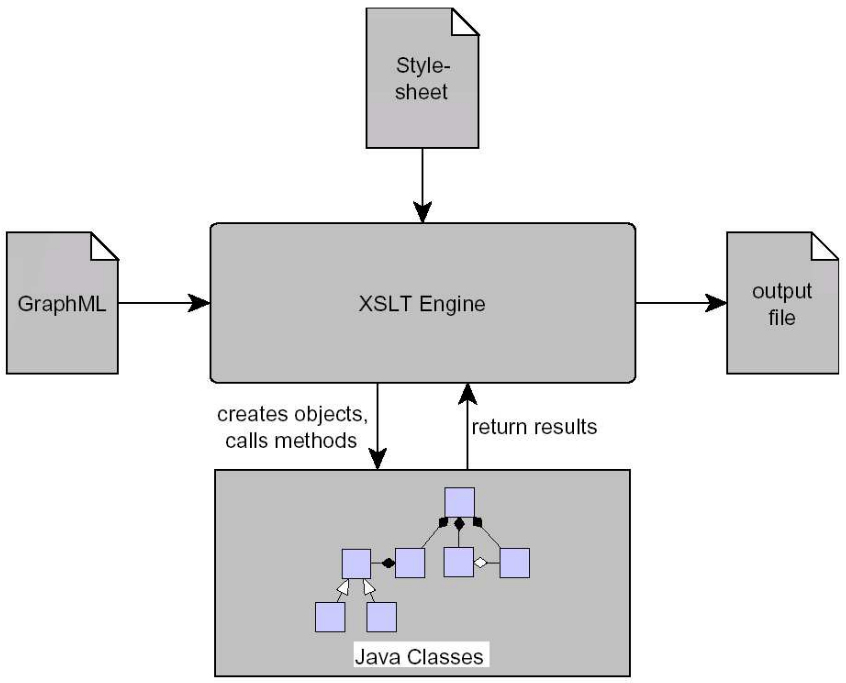

Figure 18.15 Using extension functions in XSLT. Taken from [BP04].

nates and other layout data in order for the style sheet to insert it into the XML (probably GraphML) result of the transformation, or to do further computations.

The approach presented here is only one of many ways of mapping an external graph description file to an internal graph representation. A stand-alone application could integrate a GraphML parser, build up its graph representation in memory apart from XSLT, execute a transformation, and serialize the result as GraphML output. However, the intrinsic advantage of using XSLT is that it generates output in a natural and embedded way, and that the output generation process can be customized easily.

XSL transformations are a simple, lightweight approach to processing graphs represented in GraphML. They have proven to be useful in various areas of application, when the target format of a transformation is GraphML again, or another format with a similar purpose, and the output structure does not vary too much from input.

They are even powerful enough to specify advanced transformations that go beyond mapping XML elements directly to other XML elements or other simple text units. However, advanced transformations may result in long-winded style sheets that are intricate to maintain, and most likely to be inefficient. Extension functions appear to be the natural way out of such difficulties.

We found that, as rule-of-thumb, XSLT should be used primarily to do the structural parts of a transformation, such as creating new elements or attributes, whereas specialized extensions are better for complex computations that are difficult to express or inefficient to run using pure XSLT.

18.6 Using GraphML

The easiest way to read and write GraphML files is to use a graph-processing software that can handle this format. GraphML is the principal I/O format of visone [BBB$^+$02] and of the graph editor yEd from yWorks.[1] Besides these there are several software tools or libraries that can either import or export (or both) GraphML, including Pajek [DMB05], ORA [CR04], and JUNG [OFS$^+$05]. If a customary GraphML reader has to be implemented it is convenient to make use of one of many available XML parsers and adapt it to the purpose at hand.

[1] `http://www.yworks.com/`

Bibliography

[BBB+02] Michael Baur, Marc Benkert, Ulrik Brandes, Sabine Cornelsen, Marco Gaertler, Boris Köpf, Jürgen Lerner, and Dorothea Wagner. visone - software for visual social network analysis. In *Proc. 9th Intl. Symp. Graph Drawing (GD '01)*, pages 463–464, 2002.

[BEF99] Stephen P. Borgatti, Martin G. Everett, and Linton C. Freeman. *UCINET 6.0.* Analytic Technologies, 1999.

[BLP04] Ulrik Brandes, Jürgen Lerner, and Christian Pich. GXL to GraphML and vice versa with XSLT. *Elsevier ENTCS*, 127(1), 2004.

[BMN01] 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 410–418. Springer, 2001.

[BP04] Ulrik Brandes and Christian Pich. GraphML transformation. In János Pach, editor, *Proceedings of the 11th International Symposium on Graph Drawing (GD '04)*, volume 3383 of *Lecture Notes in Computer Science*, pages 89–99. Springer, 2004.

[Bri04] Stina Bridgeman. GraphEx: An improved graph translation service. In Giuseppe Liotta, editor, *Proceedings of the 11th International Symposium on Graph Drawing (GD '03)*, volume 2912 of *Lecture Notes in Computer Science*, pages 307–313. Springer, 2004.

[CR04] Kathleen Carley and Jeffrey Reminga. ORA: Organization risk analyzer. Technical Report CMU-ISRI-04-106, Carnegie Mellon University, 2004.

[DBETT99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, 1999.

[DMB05] Wouter De Nooy, Andrej Mrvar, and Vladimir Batagelj. *Exploratory social network analysis with Pajek.* Cambridge University Press, 2005.

[GML] GML. *The Graph Modeling Language File Format.* `http://www.infosun.fmi.uni-passau.de/Graphlet/GML/`.

[OFS+05] Joshua O'Madadhain, Danyel Fisher, Padhraic Smyth, Scott White, and Yan-Biao Boey. Analysis and visualization of network data using JUNG. *Journal of Statistical Software*, 2005.

[Sax] Saxon Open Source Project. *Saxon home page.* `http://saxon.sourceforge.net/`.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, February 1981.

[TRC03] Max Tsvetovat, Jeffrey Reminga, and Kathleen Carley. Dynetml: Interchange format for rich social network data. In *NAACSOS Conference, Pittsburgh, PA*, 2003.

[W3Ca] W3C. *Scalable Vector Graphics.* `http://www.w3.org/TR/SVG/`.

[W3Cb] W3C. *SOAP.* `http://www.w3.org/TR/soap12-part0/`.

[W3Cc] W3C. *XSL Transformations.* `http://www.w3.org/TR/xslt/`.

[Win02] Andreas Winter. Exchanging Graphs with GXL. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Proceedings of the 9th International*

Symposium on Graph Drawing (GD '01), volume 2265 of *Lecture Notes in Computer Science*, pages 485–500. Springer, 2002.