# GraphML Transformation[⋆]

Ulrik Brandes and Christian Pich[⋆⋆]

Department of Computer & Information Science, University of Konstanz, Germany
{Ulrik.Brandes,Christian.Pich}@uni-konstanz.de

**Abstract.** The efforts put into XML-related technologies have exciting consequences for XML-based graph data formats such as GraphML. We here give a systematic overview of the possibilities offered by XSLT style sheets for processing graph data, and illustrate that many basic tasks required for tools used in graph drawing can be implemented by means of style sheets, which are convenient to use, portable, and easy to customize.

## 1 Introduction

Among the multitude of software packages that process graphs, some are dedicated graph packages while others operate on graph structures implicitly. All of them have in common the need to input existing data and to output their computation results in files or streams. GraphML (Graph Markup Language) is an XML-based format for the description of graph structures, designed to improve tool interoperability and reduce communication overhead [1]. It is open to user-defined extensions for application-specific data. Thanks to its XML syntax, GraphML-aware applications can take advantage of a growing number of XML-related technologies and tools, such as parsers and validators.

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) [7] offer a more natural way of utilizing XML formatted 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.

This article is organized as follows. Section 2 provides some background on GraphML, XSLT and their combination. Basic means and concepts of transformations are outlined in Sect. 3, while different types of transformations are discussed in Sect. 4. The integration of XSLT extension mechanisms is described in Sect. 5, and results are discussed and summarized in Section 6.

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

[⋆⋆] Corresponding author.

## 2 Background

A key feature of GraphML is the separation into structural and data layer, both conceptually and syntactically; this enables applications to extend the standard GraphML vocabulary by custom data labels that are transparent to other applications not aware of the extension. Furthermore, applications are free to ignore unknown concepts appearing in the structural layer, such as `<port>`s, `<hyperedge>`s or nested `<graph>`s.

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, such as Scalable Vector Graphics [5] 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 [6].

Since GraphML representations of graphs often need to be preprocessed or converted to other XML formats, it is convenient to transform them using XSLT, a language specifically designed for transforming XML documents; while originally created for formatting and presenting XML data, usually with HTML, it also allows general restructuring, analysis, and evaluation of XML documents. To reduce parsing overhead and to allow for XML output generation in a natural and embedded way, XSLT itself is in XML syntax.

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. 1 shows the basic workflow of a transformation.

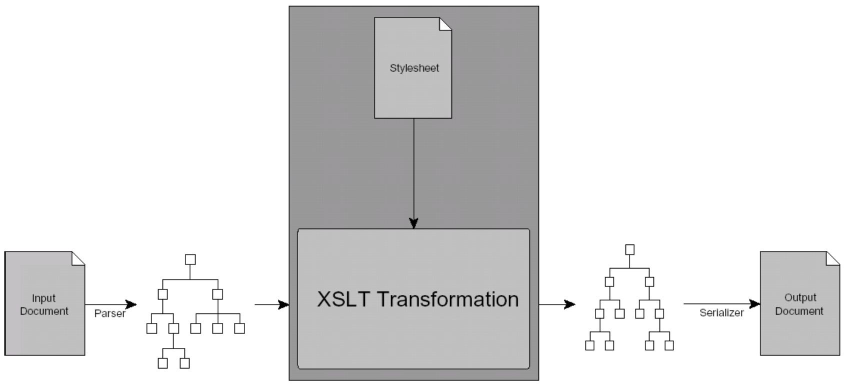

**Fig. 1.** Workflow of an XSLT transformation. First, XML data is converted to a tree representation, which is then used to build the result tree as specified in the style sheet. Eventually, the result tree is serialized as XML.

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.

## 3 Basic Means of Transformation

The expressivity and usefulness of XSLT transformations goes beyond their original purpose of only "adding 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. For some example style sheets, see Sect. 4.

### 3.1 Parameterization

Especially when integrated as component of a larger system, it is desirable or necessary to parameterize the transformations. Therefore, style sheets can be given an arbitrary number of global parameters `<xsl:param>` that serve as an interface to the outside world. When used autonomously, parameters are passed to the processor as command line parameters.

Such parameters are often used to determine which part of the source document is to be processed. For example, a GraphML file might contain multiple `<graph>`s; a parameter can express the unique identifier of a particular graph that is to be selected. Newer versions of XSLT even allow passing complex XML subtree structures to the transformation.

### 3.2 Recursion

In the pattern matching process described in Sect. 2, templates were instantiated and executed implicitly or explicitly, when a matching DOM node was encountered in the tree traversal. However, templates can also be given unique names and called like functions together with arbitrary scalar or complex arguments, independently from the tree traversal.

For implementation of more advanced computations, such as graph algorithms, templates may recursively call themselves, typically passing local parameters as function arguments. Similar to the global parameters (see Sect. 3.1), local parameters can represent both scalar values and complex tree fragments. With some limitations, XSLT can be considered a functional language, since templates (and the style sheet as a whole) define functions that are applied to subtrees of the input document and return fragments of the output document.

Due to the lack of assignment statements and side-effects, conventional imperative graph algorithms have to be formulated solely in terms of functions; states, data structures, and intermediate results must be expressed as parameters of function calls. For example, in a breadth-first search the set of all unvisited nodes is passed to a recursive incarnation of the BFS template, instead of becoming marked (see Sect. 4.3).

## 3.3 Modularization

To make transformations more flexible, they are not necessarily defined in one single file, but can be distributed over a set of modules. The main style sheet imports all templates from another style sheet with `<xsl:import>`, with its own templates having a higher priority, or includes them textually using an `<xsl:include>` tag. Alternatively, style sheets can be composed in advance instead of being imported and included at transformation runtime. Since XSLT is XML, it is even possible for style sheets to compose and transform other style sheets.

Another way of modularizing large transformations is to split them up into several smaller exchangeable style sheets that define successive steps of the transformation, each of which operates on the GraphML result produced in the previous step.

In effect, modularizing transformations facilitates implementing a family of general-purpose and specialized style sheets. Users are free to use specialized modules, or to design new custom templates that extend the general ones.

## 3.4 External Code

XSLT is designed to be an open, extensible system. While parameterization is one way of using an interface to the outside world when XSLT serves as a component, another even more powerful mechanism is the integration of extension functions into XSLT, i.e. code external to the style sheet. This is especially useful when pure XSLT implementations are inefficient to run or too complicated to use, especially when the input document is large, or when XSLT does not provide necessary functionality at all, e.g. when random numbers, mathematical operations, date functions, or complex string manipulations are needed.

It is important to note that extension functions and classes may violate the declarative, functional design idea of XSLT, since instance-level methods can provide information about mutable states, thus making side-effects possible because a template may now produce different output at different times of execution.

The mechanism is described in more detail in Sect. 5, where we present an extension to be used with GraphML.

## 4 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 transformations may correspond to more than one type.

### 4.1 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.

For such GraphML-to-GraphML transformations that operate on the syntactical representation rather than on the underlying combinatorial structure, XSLT style sheets are a very useful and lightweight tool. Often, the source code fits on one single page. See, e.g., Fig. 2 and Fig. 3.

### 4.2 Format Conversion

Although in recent years GraphML and similar formats like GXL [9] became 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.

```
<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>
```

**Fig. 2.** This transformation rearranges the graph so that the nodes appear before the edges. All subtrees related to data extensions (data and key tags) are omitted.

```
<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:import href="rearrange.xsl"/> <!-- import templates -->

  <xsl:template match="graph">
    <graph>
      <xsl:copy-of select="key|desc|@*"/>
      <xsl:copy-of select="node"/>
      <xsl:apply-templates match="edge"/>
    </graph>
  </xsl:template>

  <xsl:template match="edge"> <!-- new template rule for edges-->
    <xsl:copy>
      <xsl:copy-of select="@*[name()!='id']|*"/>
      <xsl:attribute name="id">      <!-- create new ID attribute -->
        <xsl:value-of select="generate-id()"/> <!-- XPath-generated ID -->
      </xsl:attribute>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

**Fig. 3.** The transformation in Fig. 2 is extended by importing its templates and overriding the template for graphs, as described in Sect. 3.3. Edges are copied to the output document, except for their identifiers, which are generated anew.

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 and can thus be utilized in translation services like GraphEx [3]. Converting between GraphML and GXL is discussed in [2].
- *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. In fact, this is the most natural use of style sheets, and we give an example tranformation to SVG (see Appendix A).

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.

### 4.3 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 outlined in Sect. 3.2, 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. See Fig. 4 and Appendix A.

## 5 Java 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.

```
<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:param name="source">s</xsl:param>  <!-- global parameter -->

  <xsl:template match="data|desc|key"/>

  <xsl:template match="/graphml/graph">
    <graphml>
      <graph>
        <xsl:copy-of select="@*|*[name()!='node']"/>
        <key for="node" name="distance"/>
        <xsl:variable name="bfsnodes">
          <xsl:call-template name="bfs">
            <xsl:with-param name="V" select="node[@id!=$source]"/>
            <xsl:with-param name="W" select="node[@id=$source]"/>
            <xsl:with-param name="dist" select="number(0)"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:copy-of select="$bfsnodes/node"/>
        <xsl:for-each select="node[not(@id=$bfsnodes/node/@id)]">
          <xsl:copy>
            <xsl:copy-of select="*|@*"/>
            <data key="distance">-1</data>  <!-- not reachable -->
          </xsl:copy>
        </xsl:for-each>
      </graph>
    </graphml>
  </xsl:template>

  <xsl:template name="bfs">
    <xsl:param name="dist"/> <!-- current distance to source -->
    <xsl:param name="V"/>    <!-- unvisited nodes -->
    <xsl:param name="W"/>    <!-- BFS front nodes -->
    <xsl:for-each select="$W">
      <xsl:copy>
        <xsl:copy-of select="*|@*"/>
        <data key="distance"><xsl:value-of select="$dist"/></data>
      </xsl:copy>
    </xsl:for-each>
    <xsl:variable name="new" select="$V[@id=../edge[@source=$W/@id]/@target]"/>
    <xsl:if test="$new"> <!-- newly visited nodes? -->
      <xsl:call-template name="bfs"> <!-- start BFS from them -->
        <xsl:with-param name="V" select="$V[count(.|$new)!=count($new)]"/>
        <xsl:with-param name="W" select="$new"/>
        <xsl:with-param name="dist" select="$dist+1"/>
      </xsl:call-template>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

**Fig. 4.** An algorithmic style sheet that starts a breath-first search from a source specified in a global parameter. The computed distances from that source node are attached to the nodes as data tags with a newly introduced key.

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 [4] 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.

Fig. 5 shows the architecture of a transformation integrating external classes. See Appendix A for a style sheet that makes use of extension functions for random graph generation.

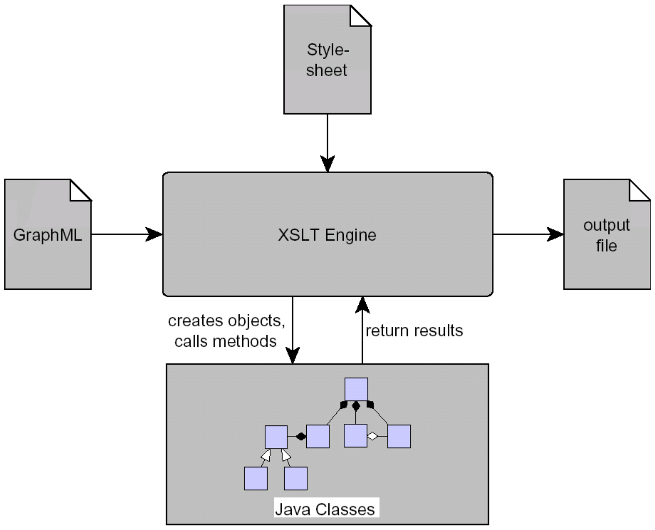

**Fig. 5.** Extending a transformation with Extension Functions. The box around the Java classes may represent a wrapper class.

In particular, this technique enables developers to implement extensions for graph algorithms. They can either implement extension functions from scratch, or make use of already existing off-the-shelf graph libraries. We implemented a prototype extension for GraphML that basically consists of three layers:

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

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 coordinates 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.

## 6 Discussion

We have presented a simple, lightweight approach to processing graphs represented in GraphML. XSLT style sheets have proven to be useful in various areas of application, both when the target format of a transformation is GraphML, and in other formats with a similar purpose where the structure of the output does not vary too much from the 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.

## 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)*, LNCS 2265:501–512. Springer, 2001.
2. U. Brandes, J. Lerner, and C. Pich: GXL to GraphML and vice versa with XSLT. *Proc. 2nd Intl. Workshop Graph-Based Tools (GraBaTs '04)*. To appear.
3. S. Bridgeman. GraphEx: An improved graph translation service. *Proc. 11th Intl. Symp. Graph Drawing (GD '03)*, LNCS 2912:307–313. Springer, 2004.
4. M. Kay. *SAXON*. http://saxon.sourceforge.net/.
5. W3C. *Scalable Vector Graphics*. http://www.w3.org/TR/SVG.
6. W3C. *SOAP*. http://www.w3.org/TR/soap12-part0/.
7. W3C. *XSL Transformations*. http://www.w3.org/TR/xslt.
8. R. Wiese, M. Eiglsperger, and M. Kaufmann. yFiles: Visualization and automatic layout of graphs. *Proc. 9th Intl. Symp. Graph Drawing (GD '01)*, LNCS 2265:453–454. Springer, 2001.
9. A. Winter. Exchanging Graphs with GXL. *Proc. 9th Intl. Symp. Graph Drawing (GD '01)*, LNCS 2265:485–500. Springer, 2001.

## A Supplement

Due to space limitations, we do not give extensive examples for style sheets. The following examples are referred to in this paper and can be obtained from the GraphML homepage (graphml.graphdrawing.org):

**GraphML → SVG.** An example for conversion into an XML-based graphics format (requires coordinates).

**Spring Embedder.** A computational style sheet computing coordinates using a popular layout algorithm.

**Random Graph Generator.** Generates random graphs in the Erdős-Rényi model by calling an external random number generator (Java language binding).