2004-08-02

Network and Graph Markup Language (NaGML) Data File Formats (extended version)

Bradley, Gordon H.

INFORMS

Bradley, Gordon H. "NETWORK AND GRAPH MARKUP LANGUAGE (NaGML)-DATA FILE FORMATS (extended version) (2004)." To appear in Proceedings of the Ninth INFORMS Computing Society Conference, 2005.
http://hdl.handle.net/10945/38179

This is an extended version of a paper that will appear in the Proceedings of the Ninth INFORMS Computing Society Conference, Annapolis, Maryland, January 5-7, 2005.

NETWORK AND GRAPH MARKUP LANGUAGE (NaGML) - DATA FILE FORMATS (extended version)

Gordon H. Bradley
Operations Research Department, Naval Postgraduate School, Monterey, CA 93943,
bradley@nps.edu
August 2, 2004

Abstract: The Network and Graph Markup Language (NaGML) is a family of Extensible Markup Language (xml) languages for network and graph data files. The topology, node properties, and arc properties are validated against the user's specification for the data values. NaGML is part of a component architecture that reads, validates, processes, displays, and writes network and graph data. Because it implements a family rather than a single xml language, NaGML offers (1) flexibility in choosing property names, data types, and restrictions, (2) strong validation, and (3) a variety of data file formats. This paper demonstrates these points with a sampling of the possible data file formats.

.

Key words: networks; graphs; Extensible Markup Language; xml; XML Schema; open source; data file format.

1. INTRODUCTION

The Network and Graph Markup Language (NaGML) is the centerpiece of an open source software project, the Network and Graph Project (Bradley, 2004a), that is constructing a suite of tools to read, validate, process, display, and write networks and graphs. NaGML is a family of xml languages to represent the topology (nodes, arcs, node sets, arc sets, and subgraphs), node properties, and arc properties. The author of a data file specifies the name and data type for each property as well as additional restrictions on the data values. The topology and values of the node and arc properties are validated using XML Schema technology. The NaGML family supports a variety of data file formats that correspond to common data formats for networks and graphs.

Figures 1, 3, 5, 7, 9, 10, and 12 each show a different network and each is written using a different NaGML language. The specific NaGML language is specified in a description that appears in the first few lines of each figure. These data files (and all other NaGML data files) are processed by a single program that uses the description in each file to automatically construct a XML Schema for the file and then use it to validate the network topology and the property values in the file. Thus NaGML users have the full power of XML Schema validation without mastering the (complex) task of constructing a comprehensive schema for their data. The description of the NaGML language in each file is used by a program that reads the data and constructs data structures to hold the network topology and property values. This paper introduces the NaGML languages and presents a sampling of the possible data file formats.

NaGML is intended to support the widest possible audience of people who use networks and graphs as well as people who construct algorithms for them. The NaGML system has simple entry points that help the casual user construct and manipulate small instances. It is also scales well for the large instances that may be constructed and consumed by production programs. This surprising capability to support a wide range of users with a variety of requirements is based on the flexibility of NaGML. NaGML achieves this by being not a single xml language, but rather a family of xml languages each with its own XML Schema whose construction is hidden from the user. Each user employs the capabilities of a custom-built xml language with a schema that is automatically constructed and applied.

The Network and Graph Project software has a component architecture that includes separate programs to read data files, validate the topology and properties, construct internal data stores, execute algorithms, display static and dynamic views, link to external systems (for example, spreadsheets and databases), and construct data files. The architecture is “loosely coupled” in the sense that it consists of components with well-defined interfaces that allow multiple implementations of each component. This allows users to construct a system from components that best meets their needs and it allows contributors to the Network and Graph Project to construct their own components that work with other components in the system.

Relational databases and SQL are the de facto standards for data storage and data access; xml has become the de facto standard for sharing data among applications. Xml is preferred for data exchange among organizations because providing xml files is preferable to allowing direct access to databases. Also since xml is character based (rather than binary) it is platform and operating system independent and thus ideal for transmission over the Internet. Recently xml has been introduced into document and spreadsheet programs to allow interoperability of content. For example, see (Goldfarb, Walmsley, 2004). There has been significant interest in developing xml languages for many data domains (Hunter, 2002; Ray, 2001).

NaGML has been designed to support the vigorous and diverse community of people who use networks and graphs for many purposes in a variety of contexts. Applications involve problem instances that range from a handful of nodes to ones with thousands and even millions of nodes and arcs. Network and graph instances are constructed to model, analyze, solve, design, display, and entertain. For example, mathematicians and social scientists analyze structure, operations researchers and computer scientists compute optimal flows and efficient structures, engineers design roads and computer chips, planners develop land use, contractors schedule projects, graphic artists develop static and dynamic displays, scientists map molecules and solar systems, and intelligence analysts try to “connect the dots.”

2. NETWORKS AND GRAPHS

A graph is a non-empty set of nodes, together with a set of arcs that are defined as pairs of nodes. The arcs can be directed (one node is the tail, the other the head) or undirected. A network is a graph with one or more properties associated with each node and arc. Each property has a fixed data type; there is a wide range of possible data types, for example, integer, double, boolean, string, etc. Nodes might have properties such as

location (x-y or lat-long), cost, and description. Arc properties might be length, name, cost, open-shut, etc.

Common examples are road and street networks, water and power grids, and communications networks. In addition to these obvious physical networks, networks (and sometimes graphs) model a wide variety of other applications such as assigning people to jobs, scheduling, bid evaluation, organization charts, and circuit board layout. Virtually all the applications envisioned for NaGML have node or arc properties (or both) and are thus networks rather than graphs; the "and graphs" was added to distinguish NaGML from markup for other networks that do not have nodes and arcs. In the subsequent discussion we drop the "and graphs."

Network data files are often input for a variety of algorithms that construct shortest paths, determine the flow of goods, schedule activities, design chips, etc. Some applications use networks as a data model to structure data. These applications use networks to store, and perhaps visualize, data. Some applications may involve only nodes and node properties; for example, data about locations (nodes) displayed on a map. NaGML capabilities support these diverse applications.

3. INTRODUCTION TO XML AND NaGML

Xml is a metalanguage for defining xml documents. It is not a language itself; instead it is a set of rules to define and construct xml documents. Markup is text that is added to a document to add meaning and structure that can then be used to automate processing of the document. This modest description hides the full range of capabilities that have been built around and on top of xml and that have lead to the rapid and widespread use of xml and xml-related technologies. See (Hunter, 2002; Ray, 2001) for a discussion of xml, see (Bradley, 2003a; Bradley, 2003b) for a discussion with operations research examples.

Xml is an open source standard that was developed by, and is supported by, the World Wide Web Consortium (W3C) see (World Wide Web Consortium). It is platform independent. There are a number of free, open source, and proprietary tools available for the efficient construction and processing of xml documents.

As shown in Figure 1, xml markup consists of elements and attributes. The elements are enclosed by a start tag: <Node nodeID="1"> that begins with the name of the element and optionally includes name-value pairs called attributes. Each element must be closed with a matching end tag: </Node>. Elements can include other elements and text (also called PCDATA). The text of an element is everything between the start tag and the end tag that is not enclosed in another element. NaGML follows common practice that limits an element to contain other elements or text, but not both. Elements are fully nested in that any start tag must be matched with its end tag inside any enclosing element. An element may also be an "empty" element that combines the start and end tags: <Arc tail ="2" head="1"/>

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Network and Graph Markup Language (NaGML) -->
<NaGXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <Description>
      <Network name="Figure 1" arcType="directed" dataFormat="general" validate="true"/>
      <NodeID nodeIDDataType="xs:integer" declared="Nodes only"/>
      <ArcID arcIDDataType="xs:integer" declared="automatic"/>
      <NodeProperties>
         <NodeProperty name="Xpixel" dataType="xs:nonNegativeInteger"/>
         <NodeProperty name="Ypixel" dataType="xs:nonNegativeInteger"/>
         <NodeProperty name="Symbol" dataType="xs:string"/>
      </NodeProperties>
      <ArcProperties>
         <ArcProperty name="Length" dataType=”xs:double”/>
      </ArcProperties>
   </Description>
   <Data>
      <Node nodeID="1">
         <Xpixel>100</Xpixel>
         <Ypixel>100</Ypixel>
         <Symbol>black circle</Symbol>
      </Node>
      <Node nodeID="2">
         <Xpixel>100</Xpixel>
         <Ypixel>200</Ypixel>
         <Symbol>red  circle</Symbol>
      </Node>
      <Node nodeID="15">
         <Xpixel>300</Xpixel>
         <Ypixel>200</Ypixel>
         <Symbol>blue square</Symbol>
      </Node>
      <Node nodeID="4">
         <Xpixel>350</Xpixel>
         <Ypixel>125</Ypixel>
         <Symbol>blue square</Symbol>
      </Node>
      <Arc tail ="4" head="4">
        <Length>34.5</Length>
      </Arc>
      <Arc tail ="2" head="4">
         <Length>160.0</Length>
      </Arc>
      <Arc tail ="15" head="4">
         <Length>200.0</Length>
      </Arc>
      <Arc tail ="2" head="1"/>
      <Arc tail ="1" head="4">
         <Length>120.4</Length>
      </Arc>
   </Data>
</NaGXML>
```

Figure 1. Simple data file format that uses defaults for node and arc identification.

Each xml document must have a unique “root” element. An xml document is a rooted tree that is almost always written in the indented format of Figure 1. By convention, element names begin with upper case letters. The name of an attribute is always followed by an equal sign and the value of the attribute in double quotes: name="Length" dataType="xs:double". We follow the convention that attribute names begin with lower case letters; in the text of this paper we will italicize attribute names and element names to make them stand out.

Since the network and graph community is international in scope, and since a data file format standard should be constructed to survive many years, it is appropriate that NaGML use xml and fully embrace the internationalization that choice allows. Xml is not “ANSI-centric” nor is it “English-centric.” It is character based (as opposed to binary) and contains full support for all character sets and all human languages. In addition to support for English, there are over 35 different encodings. The practical reality is that today (2004) few of the 35+ encodings that have been designed have been implemented into working code. However, by providing for multiple encodings, xml has laid the foundation for full internationalization, and thus made it likely that xml will remain a dominant data store format for decades, if not centuries.

An xml language (also called a tag set, an xml vocabulary, document type, xml dialect) is a specification for a set of xml documents that places restrictions on the names of elements and attributes, the structure of the elements, and the values of the data. Each xml language should have a schema that is a formal specification of these restrictions. An xml document is an instance of a particular xml language if it conforms to the schema that defines the language; this is determined by validating parser (Duckett et al., 2001; Hunter et al., 2001; van der Vlist, 2002).

There are several different kinds of schemas associated with xml (DTD, XML Schema, RELAX NG, etc.) (Duckett et al., 2001; van der Vlist, 2002). The most widely used of the comprehensive schemas (this excludes DTD) is the XML Schema defined by the W3C (World Wide Web Consortium). The schema is distinguished from its competitors by the capitalization of the X, M, L, and S characters. Every XML Schema document is also an xml document. Each kind of schema has its own characteristics that influence the definition of an xml language and determine what errors are found by validation. Here we discuss XML Schema exclusively. XML Schema is good for expressing data types for element and attribute data values. XML Schema provides 44 built-in data types that cover numbers, strings, boolean, time, dates, etc. In the figures the data types all have the prefix “xs:”. This indicates the Namespace associated with XML Schema; the prefix will be omitted when discussing the data types in the text. Data values can be further constrained with restrictions to these types, by applying patterns expressed as regular expressions, or by enumerating choices. The figures below give some idea of the range and detail that is possible. XML Schema is a so-called “closed” schema in which the structure of the document and the name of all elements must be included in the schema. A NaGML user selects the names, data types, and restrictions for their data (and thus describes an xml language for their data files) and a NaGSchemaConstruct component automatically constructs an XML Schema to validate the data.

Defining a new xml language is a difficult task that often involves some serious trade-offs. One goal is to make the language comprehensible and flexible, so that as many

people as possible adopt it as a standard. Wide adoption supports interoperability of data and can lead to the development of associated software to read, write, process, transform, and visualize data files. Another goal is to make the xml language extensible, so that it can accommodate evolving user requirements. The final goal is that the schema be detailed and comprehensive, so that it enforces strong validation on xml documents.

New and emerging requirements for interoperability of valid data (particularly across the Web) place a new set of demands on network and graph data representations. Fast, high-volume exchange of data between different organizations and tightly coupled applications that do not have humans in the process must be concerned that data file errors (in structure and in data values) do not corrupt downstream processing. Thus, each data producer and consumer must guard against a variety of data errors. Much of this responsibility can be shifted to xml validating parsers and thus greatly reduce the amount of error checking that must be included in application software. The producer of data can validate a data file before sending it to someone or some application for further processing. The validation by the data producer helps identify data errors as the data is produced—this is the time and place where it is most effective to correct errors. Consumers of validated data know that the data file conforms to the schema and thus does not contain certain errors in structure and content.

The schema defines the structure and the content of the data file and identifies as errors only deviations from this. The “strength” of a validation is a measure of the number, scope, and kinds of errors that can be identified. The construction of the schema determines which errors will be caught by validation. For example, a data item that should be “A123” could be validated to be a string (and thus, “this is data” would be valid), or a string with no spaces and beginning with a letter, or as an English upper case letter followed by exactly three digits, etc. The figures below show some of the validation that XML Schema supports. The construction of a schema for an xml language can be a complex engineering task with critical issues such as how much detail to include—more detail identifies more possible errors, but simultaneously reduces the number of documents that conform to the schema. Validation cannot catch all data errors (for example, entering “3187” instead of “3817”), but many errors can be caught and using validating parsers to check the data is preferable to writing application-specific code.

The NaGML design philosophy has been to make schemas as “strong” as possible and thus include as many checks as possible in the validation. This minimizes the error checking that a NaGReader must do. The reference implementation of the NaGSchemaConstruct has achieved this. However, validation for large data files can be demanding on computer time and space resources. It is anticipated that, in particular situations, some of these errors may not be possible given how the data is constructed, thus it is unnecessary to check for them during validation. For some applications (particularly those with larger data files) it may be efficient to move some error checking from the validation to the reader. Also, the effort to validate a data file can be reduced if the user selects only certain data formats.

4. NETWORK AND GRAPH PROJECT LOOSELY COUPLED COMPONENTS ARCHITECTURE

From its inception, NaGML has been much broader than a markup language. The Network and Graph Project includes a design for a comprehensive software system for networks that supports the full spectrum of processing including: construct, read, validate, store, solve, display, link, and write. This includes multiple data structures so that an algorithm can be paired with a data structure that best supports its calculations, visualization of both static structures and dynamic processes so that algorithms can be animated and analyzed, and linking to external systems for display and computation. The system is not "read data, apply algorithm, print solution," instead, the viewpoint is that the network data model is central and read, validate, solve, display, link, and write are just operators that transform the data.

The Network and Graph Project has a component architecture to support the operations mentioned above. The components have well-defined interfaces and are "loosely coupled" in the sense that the interfaces are minimal and abstract, and thus encourage the development of multiple implementations for each component.

The architecture of the Network and Graph Project presumes there will be multiple implementations for each component. This encourages the development of an open source community, where sharing of components allows users to select the combination of components that best supports their application. This also allows individuals who want to develop a new algorithm, innovative data structure, visualization tool, or analysis technique to concentrate on their interest, while using components constructed by others. This component architecture supports innovation by allowing easy access to, and testing of, new components.

One important activity of an open source component architecture project is to construct reference implementations that demonstrate that components that satisfy the interface can be effectively constructed. Currently (summer 2004), there are reference implementations for two NaGReaders, a NaGSchemaConstruct, a hash table based data store, a visualization tool to construct tables, a visualization system to animate time series data over a map (the application mentioned below), and a NaGWriter that constructs NaGML data files and comma-separated output to link to an Excel spreadsheet. NaGReaders access a data file and, guided by information in the file, optionally invoke another program that constructs a schema (XML Schema) for the file; optionally validates the file using a schema; and always processes the file to construct a "dataStore" that is the internal representation of the network. Following xml practice, if the data file fails validation, it should not be processed. The details of the flexible data formats that NaGML allows do not persist into the dataStore, thus an application can select from various data format/NaGReader combinations to construct the same dataStore, which then interacts with the other components.

5. NaGML AS A PERSISTENT DATA MODEL

NaGML is a data model for representing data about points and relationships between pairs of points. While this is by no means a model of everything, it is clearly more than a data file format.

Networks and graphs have the interesting property that intermediate calculations and algorithm results are most often also node and arc properties. Thus, the data model is not just for input; it persists through calculations that modify and add properties. This suggests a data-centric view, where algorithms are just operators on the data that add and modify properties, while the network structure persists. This perspective is particularly valuable when an application or process consists of a sequence of algorithms. Another example of the persistence of the data model is that the software to display data files can also be used for static or dynamic views of intermediate calculations and for presentations of results.

The network data model is even valuable in situations where network terminology is not used. As noted below, the flexibility of NaGML can bring network and graph tools to other applications, where users have the notion of points and lines, even if they don't use the terminology of networks, graphs, nodes, and arcs.

6. DATA FORMATS

One of the goals of NaGML is to provide flexible data formats that support the full range of common network data formats. The focus of this paper is to demonstrate the variety of data file formats that are supported and the data validation that is provided.

The first thing to notice about the examples is that the user chooses the names for node and arc properties. This seems like a fairly basic requirement, but in fact, the design that permits this is the most innovative part of the NaGML design. XML Schema supports the development of a range of different languages, but in each specific language the names of the elements are fixed. The NaGML mechanism to allow users to select their own names for node and arc properties, and to specify data types and further restrictions for data values, is described in a companion paper (Bradley, 2004b) written for an xml audience. It is not exaggeration to say that the Network and Graph Project would not have been possible without this innovation to common xml practice.

The second thing to notice is the wide variety of data types that are supported. While it might seem sufficient for most operations research applications to have only integer and double properties, the extension to strings, booleans, times, dates, lat-long, URLs, etc. is essential for the full range of network applications. As mentioned below, the first application of NaGML was an operations research analysis where only a few of the properties are integer or double. Finally note the quite different ways the data files can be defined.

The first example is shown in Figure 1. The *Description* element contains the specification of each property. The NaGReader reads the data file and invokes a NaGSchemaConstruct program to construct the appropriate schema and then validates the topology and property data values. It then constructs a dataStore that contains the topology, properties, and data type information. Figure 2 shows a table view of the arc properties. The table is constructed from information in the dataStore. Note that by user option, missing data values are displayed as blanks (rather than 0 or 0.0). The user can select any column to sort the rows.

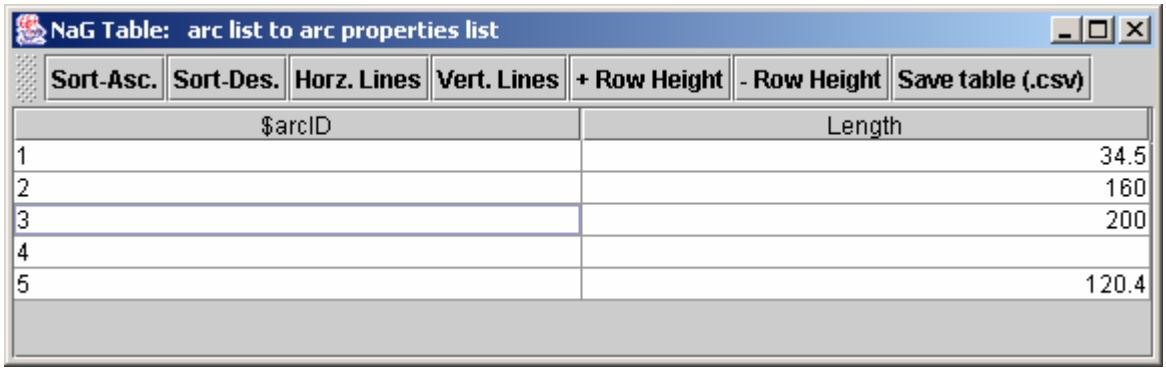

Figure 2. Table view of the arc property values from the Figure 1 data file.

Each node and each arc has a unique identifier, *nodeID* and *arcID*. In Figure 1, the defaults are used; *nodeID* is an integer and is assigned in each *Node* element. The *nodeID* values need not be in order in the data file or contiguous integers (or even positive). The validation process checks that each *nodeID* is a legal integer and each node has a unique value. The default for *arcID* is integers 1, 2, … assigned in the order the arc appears in the data file (called "document order"). The validation checks that each head and tail attribute is assigned to one of the nodes declared in the data file. The validation process checks that each *Xpixel* and *Ypixel* data value is a non-negative integer and each *Length* is a legal double value.

This format shows that with only a small amount of training a user can quickly construct a small network. The plans for the Network and Graph Project include having input from spreadsheets, databases, forms, and Web browsers that will allow even simpler access to NaGML capabilities.

This data format allows any number of nodes and arcs. As shown in subsequent examples, there is no restriction on the ordering of nodes and arcs and no requirement that nodes precede arcs.

In the subsequent figures, the defaults are explicitly included in order to show where and how the parameters of the system are specified. In Figure 3, the *nodeID* is still integer, however, the user has specified the exact number of nodes and their *nodeID* by specifying the attribute *declared* = "10 to 13." The validation enforces that there will be exactly 4 nodes with the specified *nodeID*s. The *arcID* is specified to be a string and the attribute declared = "Arcs only" indicates that they will be specified in the *Arc* elements. The *arcID*s can be any string, but the string must be unique for each arc. Duplicate arcs (each with a different *arcID*) are allowed. The validation checks each property value, which now includes string, date, and boolean values. The integer node properties *Xpixel* and *Ypixel* specify pixel locations so that a NaGViewer can display the network. The non-negative integers are further restricted so the nodes will appear inside a window that is 400 by 600 pixels. All the values for node property *DateConstructed* must be correctly formatted XML Schema dates (for example, 2001-04-23). The arc property *RoadOpen* values must be boolean.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Network and Graph Markup Language (NaGML) -->
<NaGXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <Description>
      <Network name="Figure 1" arcType="directed" dataFormat="general" validate="true"/>
      <NodeID nodeIDDataType="xs:integer" declared="10 to 13"/>
      <ArcID arcIDDataType="xs:string" declared="Arcs only"/>
      <NodeProperties>
            <NodeProperty name="Xpixel" dataType="xs:nonNegativeInteger" maxInclusive="400"/>
            <NodeProperty name="Ypixel" dataType="xs:nonNegativeInteger" maxInclusive="600"/>
            <NodeProperty name="Symbol" dataType="xs:string"/>
            <NodeProperty name="DateConstructed" dataType="xs:date"/>
      </NodeProperties>
      <ArcProperties>
            <ArcProperty name="Length" dataType="xs:double"/>
            <ArcProperty name="RoadOpen" dataType="xs:boolean"/>
      </ArcProperties>
   </Description>
   <Data>
      <Arc arcID="a-23" tail ="12" head="13">
         <Length>160.0</Length>
      </Arc>
      <Node nodeID="10">
         <Xpixel>100</Xpixel>
         <DateConstructed>2001-04-23</DateConstructed>
         <Ypixel>100</Ypixel>
         <Symbol>black circle</Symbol>
      </Node>
      <Node nodeID="12"/>
      <Node nodeID="11">
         <DateConstructed>1990-10-10</DateConstructed>
         <Xpixel>300</Xpixel>
         <Ypixel>200</Ypixel>
         <Symbol>blue square</Symbol>
      </Node>
      <Arc arcID="unknown" tail ="11" head="12">
         <Length>120.4</Length>
      </Arc>
      <Node nodeID="13">
         <Xpixel>350</Xpixel>
         <Ypixel>125</Ypixel>
         <Symbol>blue square</Symbol>
         <DateConstructed>1954-01-26</DateConstructed>
      </Node>
      <Arc arcID="San Francisco" tail ="13" head="10">
         <Length>34.5</Length>
      </Arc>
      <Arc arcID="Washington" tail ="12" head="13">
         <Length>200.0</Length>
      </Arc>
      <Arc arcID="warehouse 1" tail ="12" head="11"/>
   </Data>
</NaGXML>
```

Figure 3. Data with *nodeID* 10 to 13 and *arcID* declared in each *Arc* element.

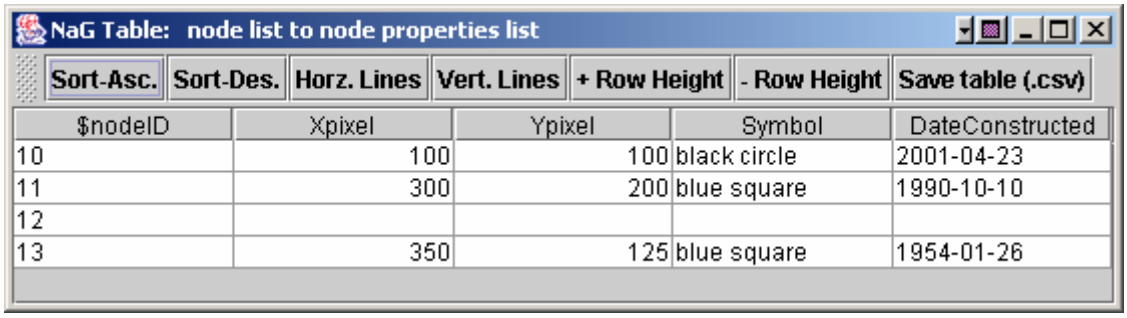

Figure 4. Table view of node property values (one of 16 views of the data).

In the previous figures there is a *Node* element for each node. Some networks have no node properties; a common data format for this lists only the arcs with the implication that there should be a node created for the tail and head nodes specified. The *NodeID declared* = "Nodes then Arcs" indicates that a node is created for each *Node* element in the data file. In addition, absent a *Node* element, any node referred to in an *Arc* element is created. This option can be used even if there are node properties; in that case, a *Node* element is required only if the node has a non-empty property.

Figure 5 also shows that it is possible to enumerate the possible values that an arc (or node) property can assume. The figure shows this for a string property but this works equally well for other data types.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Network and Graph Markup Language (NaGML) -->
<NaGXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
   <Description>
      <Network name="Figure 5" arcType="directed" dataFormat="general" validate="true"/>
      <NodeID nodeIDDataType="xs:string" declared="Nodes then Arcs"/>
      <ArcID arcIDDataType="xs:string" declared="Arcs only"/>
      <ArcProperties>
            <ArcProperty name="Length" dataType="xs:double"/>
            <ArcProperty name="RoadCondition" dataType="xs:string">
                  <Enumeration>closed</Enumeration>
                  <Enumeration>open</Enumeration>
                  <Enumeration>open if dry</Enumeration>
                  <Enumeration>unknown</Enumeration>
            </ArcProperty>
      </ArcProperties>
   </Description>
   <Data>
      <Arc arcID="I 80" tail ="Boston" head="New York">
         <Length>160.0</Length>
         <RoadCondition>open</RoadCondition>
      </Arc>
      <Arc arcID="I 40" tail ="New York" head="Chicago">
         <Length>120.4</Length>
         <RoadCondition>open if dry</RoadCondition>
      </Arc>
      <Arc arcID="I-80W" tail ="Chicago" head="Denver">
         <Length>34.5</Length>
         <RoadCondition>closed</RoadCondition>
      </Arc>
      <Arc arcID="the 101" tail ="Los Angeles" head="San Diego"/>
      <Arc arcID="the 10" tail ="Los Angeles" head="Phoenix">
         <RoadCondition>unknown</RoadCondition>
         <Length>200.0</Length>
      </Arc>
   </Data>
</NaGXML>
```

Figure 5. Nodes implicitly declared and an arc property enumerated.

NaG Table: node arc incident matrix: node list to incident arc list

Sort-Asc. | Sort-Des. | Horz. Lines | Vert. Lines | + Row Height | - Row Height | Save table (.csv)

nodeID/arcID	I 80	I 40	I-80W	the 101	the 10
Boston	1				
New York	-1	1			
Chicago		-1	1		
Denver			-1		
Los Angeles				1	1
San Diego				-1	
Phoenix					-1

Figure 6. Node-arc incident matrix (node adjacency matrix is also available).

Many network algorithms use a “forward star” data store where all the arcs with the same tail node are stored contiguously. Sometimes it is convenient to have this reflected in the data file Figure 7 shows all the data in forward star form. NaGML also supports reverse star, which is not shown. Forward star and reverse star data formats can be combined with any of the formats shown in the previous figures.

```
<?xml version="1.0" encoding="UTF-8"?>
<NaGXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <Description>
      <Network name="Figure 7" arcType="directed" dataFormat="general" validate="true"/>
      <NodeID nodeIDDataType="xs:string" declared="list">12  34  14  23  -3</NodeID>
      <ArcID arcIDDataType="xs:NMTOKEN" declared="list">
         12to34  34to14  14to23  14to23Again</ArcID>
      <NodeProperties>
            <NodeProperty name="StartTime" dataType="xs:time"/>
            <NodeProperty name="Signal" dataType="xs:integer"/>
      </NodeProperties>
      <ArcProperties>
            <ArcProperty name="Cost" dataType="xs:double"/>
      </ArcProperties>
   </Description>
   <Data>
      <Node nodeID="12">
         <StartTime>13:23:59</StartTime>
         <Signal>-345</Signal>
         <ArcTail arcID="12to34" head="34">
            <Cost> 34.56 </Cost>
         </ArcTail>
      </Node >
      <Node nodeID="14">
         <StartTime>01:45:00</StartTime>
         <Signal>0</Signal>
         <ArcTail arcID="14to23" head="23">
            <Cost> 26.99 </Cost>
         </ArcTail>
         <ArcTail arcID="14to23Again" head="23">
            <Cost> 0.00 </Cost>
         </ArcTail>
      </Node >
      <Node nodeID="34">
         <ArcTail arcID="34to14" head="14">
            <Cost> -23.00 </Cost>
         </ArcTail>
         <StartTime>23:23:23</StartTime>
         <Signal>222</Signal>
      </Node >
      <Node nodeID="-3"/>
      <Node nodeID="23">
         <StartTime>12:00:00</StartTime>
         <Signal>444</Signal>
      </Node >
   </Data>
</NaGXML>
```

Figure 7. Data file with forward star format and lists of *nodeID*s and *arcID*s.

As shown in Figure 7, the user can specify lists of *nodeID*s and *arcID*s that must be used in the *Data* element. The validation checks that these, and only these, are used in the *Data* element. Lists of values in xml must be space-separated (comma-separated lists can be used, but they are treated as a single string and thus are not subject to the validation of the individual values that we demand). This means that values in lists cannot contain leading, trailing, or embedded spaces. Thus, while "San Francisco" can be used as a *nodeID*, *arcID*, or property, it cannot be used in a list. The XML Schema data types include a restricted string type, NMTOKEN, that does not allow leading, trailing, or embedded spaces. If the user intends to forbid spaces in the *nodeID*, *arcID*, or a property, specifying this restricted data type allows the validation to enforce the no-space decision.

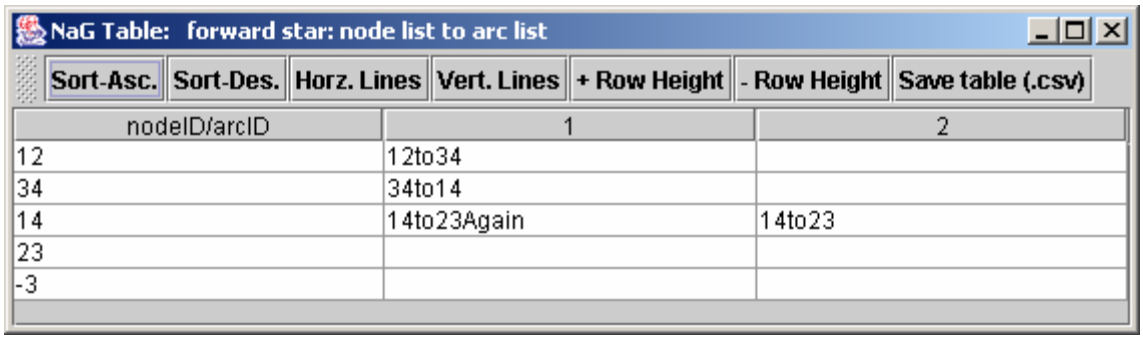

Figure 8. Table view in forward star format (reverse star is also available).

The loosely coupled components architecture guarantees that the data file format is completely independent of the dataStore, so the display of the data in the tables does not depend in any way on the format of the data file.

One of the advantages of xml is that the data is represented as character data (as opposed to binary formats that may be computer dependent) so it is "human readable." This encourages the descriptive (and often long) names for the properties that help to document the data files. In the resulting data files the ratio of markup to data values can be high. This is appropriate for files that are constructed and read by people. For large data files, and for data files that are constructed and consumed by computer programs without human interaction with the data, it is useful to reduce this markup. (Note: It is not clear that reducing the markup is necessary even for large data files because xml files can be efficiently and effectively compressed.) One mechanism is to shorten the property names as shown in Figure 9. As shown in the *NodeProperty* and *ArcProperty* elements, there is an optional attribute *label* that provides a name that can be used in generating reports or when the data is viewed by people.

There are several other ways to reduce markup. As shown for the *Node* element "123," non-empty node data can be included as attributes. This format is also a compact way to keep all the data associated with a node (or arc) together in the data file. Node property *Location* is entered so that the values of a single property are kept together in the data file. The data is in a space-separated list. The order of the data must conform to the order of the nodes in the data file. This ordering is unambiguous for each of the four different ways to assign *nodeID*s (and each of the four ways to assign *arcID*s).

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Network and Graph Markup Language (NaGML) -->
<NaGXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
   <Description>
      <Network name="Figure 9" arcType="directed" dataFormat="general" validate="true"/>
      <NodeID nodeIDDataType="xs:integer" declared="Nodes only"/>
      <ArcID arcIDDataType="xs:integer" declared="automatic"/>
      <NodeProperties>
            <NodeProperty name="T" dataType="xs:time" label="Time"/>
            <NodeProperty name="S" dataType="xs:integer" label="Special Signal"/>
            <NodeProperty name="Location" dataType="xs:string"/>
      </NodeProperties>
      <ArcProperties>
            <ArcProperty name="C" dataType="xs:double" label="Cost"/>
            <ArcProperty name="Value" dataType="xs:double" default="100.0"/>
             <ArcProperty name="AnotherValue" dataType="xs:integer" defaultInsert="0"/>
      </ArcProperties>
   </Description>
   <Data>
      <Node nodeID="123">
         <NodeValues nodeID="123" S="-345" T="13:23:59"/>
      </Node>
      <NodeListValuesAll>
            <Location> A1 A2 B1 B2 B3 C1 C4</Location>
      </NodeListValuesAll>
      <Node nodeID="12">
         <NodeValues nodeID="12" S="45" T="10:23:59"/>
      </Node>
      <Node nodeID="100">
         <NodeValues nodeID="100" T="00:00:00"/>
      </Node>
      <Node nodeID="101">
         <NodeValues nodeID="101" T="01:00:00"/>
      </Node>
      <Node nodeID="102">
         <NodeValues nodeID="102" T="02:00:00"/>
      </Node>
      <Arc head="123" tail="23">
         <C>23.45</C>
      </Arc>
      <Arc head="123" tail="23">
         <C>23.45</C>
      </Arc>
     <Arc head="100" tail="101">
        <C>00.45</C>
      </Arc>
      <Arc head="101" tail="100">
         <C>23.00</C>
         <Value>23.23</Value>
      </Arc>
      <Node nodeID="-3"/>
      <Node nodeID="23">   <T>12:00:00</T>   <S>444</S>  </Node >
   </Data>
</NaGXML>
```

Figure 9. Short property names, properties as attributes, and property lists.

Assigning default values for node or arc properties (see arc property Value) can reduce the size of the data file. The *default* value is passed to the dataStore; a NaGReader places the value into the dataStore only if the attribute *defaultInsert* is specified (see arc property AnotherValue).

Figure 10 shows the use of *NodeList* and *ArcList* elements that offer yet another way to organize the data file. This format is useful if the property values are non-empty for only a recognized subset of the data. In addition to data entry, node sets and arc sets are useful to specify a part of the structure of the network and can be used to specify subgraphs. All node and arc sets and subgraphs are carried on into the dataStore so they can be used in algorithms, displays, and output files.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Network and Graph Markup Language (NaGML) -->
<NaGXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
   <Description>
      <Network name="Figure 10" arcType="directed" dataFormat="general" validate="true"/>
      <NodeID nodeIDDataType="xs:integer" declared="Nodes only"/>
      <ArcID arcIDDataType="xs:string" declared="Arcs only"/>
      <NodeProperties>
         <NodeProperty name="Name" dataType="xs:string"/>
              <NodeProperty name="Location" dataType="xs:string"/>
              <NodeProperty name="Time" dataType="xs:integer" label="Z time"/>
              <NodeProperty name="Date" dataType="xs:string"/>
              <NodeProperty name="LatLong" dataType="xs:string"/>
              <NodeProperty name="Cost1" dataType="xs:double" label="operations cost"/>
              <NodeProperty name="Cost2" dataType="xs:double" minInclusive="4.0"
                   maxInclusive="7.0" />
              <MultipleNodeProperties name="BothCosts" dataType="xs:double">
                  Cost1 Cost2</MultipleNodeProperties>
              <MultipleNodeProperties name="RestrictedCost1" dataType="xs:double"
                  minInclusive="4.0" maxInclusive="5.0">Cost1</MultipleNodeProperties>
      </NodeProperties>
      <ArcProperties>
              <ArcProperty name="Name" dataType="xs:string"/>
              <ArcProperty name="Capacity" dataType="xs:integer" default="3"/>
      </ArcProperties>
      <NodeLists>
              <NodeList nodeListName="SomeNodes">2  4  6 </NodeList>
              <NodeList nodeListName="OtherNodes">3  5</NodeList>
      </NodeLists>
      <ArcLists>
              <ArcList arcListName="SomeArcs">A1  A3</ArcList>
              <ArcList arcListName="OtherArcs">A2   A1</ArcList>
      </ArcLists>
   </Description>
   <Notes>
          <Processing nodePropertyDuplicate="replace" arcPropertyDuplicate="replace"
            errorMessages="prompt"/>
          <Comment name="create date">12 Feb 2004</Comment>
          <Comment name="author">G. Bradley</Comment>
   </Notes>
   <Data>
      <Node nodeID="1">
              <LatLong>N36 45 12 W104 56 12</LatLong>>
              <BothCosts>4.3  5.6</BothCosts>
      </Node>
```

```
        <Node nodeID="2">
                <Date> 2004-05-10</Date>
        </Node>
        <Node nodeID="3">
                <RestrictedCost1>4.5</RestrictedCost1>
        </Node>
        <Node nodeID="4">
                <BothCosts>4.43  5.46</BothCosts>
        </Node>
        <Node nodeID="5">
                <BothCosts>14.3  35.6</BothCosts>
        </Node>
        <Node nodeID="6">
                <RestrictedCost1>4.4445</RestrictedCost1>
        </Node>
        <Arc arcID ="A1" tail="1" head="3">
           <Name>San Francisco</Name>
        </Arc>
        <Arc arcID ="A2" tail="3" head="5">
           <Name>Atlanta</Name>
        </Arc>
        <Arc arcID ="A3" tail="5" head="3">
           <Name>Dallas</Name>
        </Arc>
        <Arc arcID ="A4" tail="6" head="3"/>
        <SubGraph name="somePart" nodeListName="someNodes" arcListName="someArcs"/>
    </Data>
</NaGXML>
```

Figure 10. Node and arc lists and multiple property elements.

It is sometimes convenient to group some of the node properties into a list. This may be to reduce markup or just to keep related data values together. A *MultipleNodeProperties* is defined by a list of node properties. The basic philosophy behind NaGML is that every data value should be subject to validation and that validation should be as "strong" as possible. XML Schema validation allows only one data type for items in a list. For this reason, each *MultipleNodeProperties* must have its own data type (plus any appropriate restrictions). As shown for *MultipleNodeProperties BothCosts*, this usually means a relaxation of the data type to allow values from all the properties. However, it can also be used to further restrict a property value as shown for *MultipleNodeProperties RestrictedCost1*. The property *Cost1* can be entered in a *Cost1* element or a *RestrictedCost1* element; for the former the validation checks that the value is double, for the later additional restrictions are imposed.

The *MultipleNodeProperties* and *MultipleArcProperties* are carried on into the dataStore. One use is to restrict the data displays for networks with large numbers of properties or to construct custom displays and reports or to create new data files with a subset of the properties. Figure 11 shows the data from Figure 10 where the *NodeList SomeNodes* and the *MultipleNodeProperties BothCosts* are used to define which nodes and node properties are displayed.

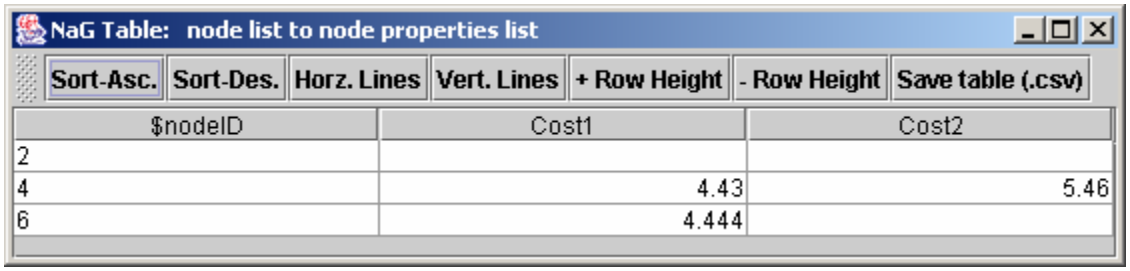

Figure 11. From Figure 10 file, demonstrates two ways to enter property data.

The validation of each data file checks the data type and restrictions for each *nodeID*, *arcID*, and property value and checks that *nodeID*s and *arcID*s are unique. However, the great variety in the ways that a data value can be included in the data file makes it impractical to check if a data value has been specified in the Data element more than once. This can be viewed as a valuable feature in that additional data can be added to the end of a (perhaps large) data file to update a value in the file. The data file author has control over what a NaGReader does whenever a duplicate property value is encountered. The *Notes* element includes an optional element *Processing* that includes directions that are passed on to a NaGReader. The user can specify that a duplicate value replaces the previous value, is ignored, or causes a fatal error. The specification for NaGReaders details the order that the elements must be processed, thus the definition of "previous" is unambiguous.

In addition to the restrictions on the data types that have been shown in the previous figures, it is also possible to specify a regular expression (using the attribute *pattern*) to further restrict the value of *nodeID*, *arcID*, and property values. XML Schema includes a powerful regular expression capability based on UNIX and Perl regular expressions ("based" means there are a few non-obvious exceptions). Regular expressions are a powerful capability to tightly specify the format and values of special data values (for example, lat-long, non-standard times and dates, phone numbers, serial numbers, etc.); however, the processing cost can be significant.

Graphs and networks have been around for hundreds of years and they are used in a wide variety of contexts. It is important for NaGML to support a wide range of data file formats in order to expedite the transition to a standard data file format. The requirement to simultaneously support all the data formats shown in the previous figures is indicated in the attribute *dataFormat* = "general" in the *Description* element. This default *dataFormat* is supported with the reference implementation reader, which uses the JDOM API (McLaughton, 2001) to access and then process the elements in a fixed sequence. JDOM constructs a tree structure for an xml document in memory. This allows "random" access to all parts of the representation of the data file, thus the *dataFormat* = "general" adds only a small additional computational cost to allow the top-level elements in the Data element to appear in any order.

For larger data files the JDOM construction is not practical; the most effective way to read the data file is in a single pass. This can be done using an xml SAX parser or by constructing a program to read the file directly (as done in the second NaGReader reference implementation). The attribute *dataFormat* in the *Description* element specifies the structure that the user has selected for the contents of the Data element. As shown in

the previous figures, the default choice of “general” offers the greatest choice of structure and order. The choice of “one pass” means that the schema that is constructed for the data file must guarantee that any NaGReader can read the data file in a single pass through the file. One pass imposes several “define before use” restrictions that must be enforced in the validation. In particular, since each arc requires a reference to a tail node and a head node that should be defined before the arc, all node elements must precede all arc elements and *ArcTail* and *ArcHead* elements are not allowed inside *Node* elements. Also the *NodeList* and *ArcList* elements (in the *Data* element) should follow the *Node* and *Arc* elements. The schema that is constructed enforces this ordering.

There are a number of choices that could be made to limit the data file format and thus allow more efficient readers. The only restriction on extending NaGML in this way is that the schema that is constructed must enforce the restrictions on structure and order, and any NaGReader should refuse to read a file that has a format it cannot correctly process.

As discussed earlier, NaGML is intended for an international community and thus has been constructed so that the visible parts of the markup (element and attribute names) can be changed to different languages (xml encodings). In the reference implementation, NaGReaders access the names of elements and attributes through a list of string constants. Thus, any change in this list changes the text of the markup without modifying any of the code. In the same way, NaGML can be modified to construct tools for communities that work with points and lines, but who have a vocabulary that does not include nodes and arcs. Figure 12 is an example of a data file format that could be supported.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Social Network for people and relationships : -->
<NaGXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <Description>
      <SocialNetwork name="Figure 12" relationships="reciprocal" dataFormat="general"
         validate="true"/>
      <PeopleNumber peopleNumberDataType="xs:integer" declared="People only"/>
      <RelationshipNumber RelationshipNumberDataType="xs:string"
            declared="Relationships only"/>
      <PeopleProperties>
            <PeopleProperty name="Name" dataType="xs:string"/>
            <PeopleProperty name="Location" dataType="xs:string"/>
            <PeopleProperty name="Age" dataType="xs:positiveInteger"/>
            <PeopleProperty name="Gender" dataType="xs:string">
               <Enumeration>male</Enumeration>
               <Enumeration>female</Enumeration>
            </PeopleProperty>
      </PeopleProperties>
      <RelationshipProperties>
            <RelationshipProperty name="Type" dataType="xs:string"/>
             <RelationshipProperty name="TimeEstablished" dataType="xs:time"/>
       </RelationshipProperties>
      <PeopleLists>
            <PeopleList PeopleListName="SomePeople">2 4 6 </PeopleList>
      </PeopleLists>
      <RelationshipLists>
            <RelationshipList RelationshipListName="SomeRelationships">A1 A3</RelationshipList>
      </RelationshipLists>
   </Description>
   <Notes>
      <Processing PeoplePropertyDuplicate="replace" RelationshipPropertyDuplicate="replace"
         errorMessages="prompt"/>
      <Comment name="create date">12 Feb 2004</Comment>
      <Comment name="author">G. Bradley</Comment>
   </Notes>
   <Data>
      <People PeopleID="1">
               <Name>John Jones</Name>>
               <Age>42</Age>
      </People>

etc...

      <Clique name="somePart" PeopleListName="somePeople"
        RelationshipListName="someRelationships"/>
   </Data>
</NaGXML>
```

Figure 12. Partial data file shows NaGML support for different element names.

7. RELATED WORK

The first application of NaGML uses many of the components described here (Schneider, 2004). It is an application to track military incidents and then to analyze them for patterns that are changing over time. Each incident (node) has 34 properties, only a few are integers or doubles. Data collection has begun and a software system to collect, visualize, and analyze the data has been deployed.

The author of a NaGML data file specifies the name, data type, and restrictions for each node and arc property. This is the most innovative feature of NaGML. Previous proposals for xml languages for networks and graphs have defined a single xml language and thus have significant limitations on what can be in a data file and/or what validation can be applied (Bradley, 2003a; Brandes, et al.; Fourer, Lopes, Martin, 2004; Holt, Schurr, Sim, Winter; Martin, 2003; Punin, Krishnamoorthy).

There are proposed xml standards for optimization that cover linear, nonlinear, integer, and stochastic programming (Fourer, Lopes, Martin, 2004; Lopes, Fourer; Martin, 2003). Network optimization problems can be modeled as more general optimization problems, however, these system do not have special markup for network problems.

There have been previous systems to support the construction and execution of graph and network algorithms (Bradley, Fernandes Oliveira, 1994). The availability of xml, and the development of NaGML, addresses many of the deficiencies in pre-xml systems.

8. NaGML—THE DESIGN CHALLENGE AND ROAD FORWARD

The adoption of new technology (xml) involves costs to the potential users of NaGML, thus the challenge is to provide sufficient benefits (for example, validation, readers, writers, connections to other data sources, displays) and simple entry (familiar, easy to learn and use data file formats). The design challenge is to address all of the sometimes conflicting design criteria (wide-spread adoption, loosely coupled components architecture, flexible data format, and strong validation) without sacrificing any one of them. The dynamic construction of NaGML schemas provides strong validation with no loss of flexibility. The other design criteria, and full details on the NaGML architecture and the reference implementation of several components, are described in detail in (Bradley, 2004a).

A further challenge for the Network and Graph Project is to evolve a component architecture that helps create an open source community that will support the independent development of components by many people.

Other data formats (in particular, databases and spreadsheets) have announced xml formats with accompanying schemas. One task of the Network and Graph Project is to replace the existing components that link to spreadsheets via comma-separated lists with connections based on NaGML and the corresponding schemas of other documents. This will add support for validation as well as provide access to the capabilities of these systems.

9. CONCLUSIONS

NaGML is a family of xml languages for network and graph data. As shown in the previous figures, NaGML offers flexibility in choosing property names, data types, and restrictions. NaGML has a simple syntax to specify the name of each property and its data type from among the 44 built into XML Schema. The data types can be modified by adding further restrictions and by applying regular expressions. NaGML offers strong validation. Using software from (Bradley, 2004a), a custom XML Schema is automatically constructed and used to validate that each data value confirms to the specified data type and restrictions. NaGML offers a variety of data file formats. As shown in Figures 1, 3, 5, 7, 9, 10, and 12, data values can be specified in a variety of ways. All the possible data file formats are read by a single software component from (Bradley, 2004a).

In addition to software components to construct schema, validate data values, and read data files, (Bradley, 2004a) has components to display network data and to write NaGML files. Future plans include components for executing network algorithms and visualizing static and dynamic views of algorithm calculations.

ACKNOWLEDGEMENT

This research has been supported by a grant from the Mathematical Sciences Division, Office of Naval Research.

REFERENCES

Bradley, G., 2003a, "Extensible Markup Language (XML) with Operations Research Examples," tutorial given at the Eighth INFORMS Computing Society Conference, January 2003, Chandler, AZ http://diana.or.nps.navy.mil/~ghbradle/xml/PaperMay2003/GBradleyXMLTutorialJan03.zip.

Bradley, G., 2003b, "Introduction to Extensible Markup Language (XML) with Operations Research Examples," INFORMS Computing Society Newsletter, Vol. 24, Number 1, Spring 2003, page 1 (14 pages). HTML version with live links: http://faculty.gsm.ucdavis.edu/~dlw/bradleyNewsletter.htm

Bradley, G., 2004a, Network and Graph Project, see http://diana.or.nps.navy.mil/~ghbradle/NetworkAndGraphProject for a description of this open source project and a link to a repository that contains the project code, examples, and documentation.

Bradley, G., 2004b, "Schema Construction for a Family of xml Languages" (in preparation).

Bradley, G. and Fernandes Oliveira, H., 1994, "NETWORK ASSISTANT to Construct, Test and Analyze Graph and Network Algorithms," in *Computational Support for Discrete Mathematics*, eds. N. Dean and G. Shannon, American Mathematical Society, 1994, pp. 75-84.

Brandes, U. Eiglsperger M., Herman I., Himsolt M., and Marshall, M., "GraphML," http://graphml.graphdrawing.org/.

Common Optimization Interface for Operations Research (COIN-OR), http://www-124.ibm.com/developerworks/opensource/coin/.

Duckett, J., et al., 2001, *Professional XML Schemas*, WROX.

Fourer, R., Lopes L., and Martin K., 2004, "LPFML: A W3C XML Schema for Linear Programming," http://gsbkip.uchicago.edu/fml/fml.html.

Goldfarb, C. F. and Walmsley P., 2004, *XML in Office 203*, Prentice Hall.

Holt, R., Schürr, A, Elliott Sim, S., and Winter A., "Graph Exchange Language," http://www.gupro.de/GXL/.

Hunter, D., et al., 2002, *Beginning XML*, 2nd edition, WROX.

Lopes, L. and Fourer R., "SNOML," http://senna.iems.nwu.edu/xml/.

Martin, K., 2002, "A Modeling System for Mixed Integer Linear Programming Using XML Technologies," December 11, 2002, revised February 27, 2003, 34 pages. http://gsbkip.uchicago.edu/xslt/pdf/xmlmodeling.pdf.

McLaughton, B., 2001, *Java & XML*, 2nd edition, O'Reilly.

Punin, J. and Krishnamoorthy M., "XGMML (eXtensible Graph Markup and Modeling Language)," http://www.cs.rpi.edu/~puninj/XGMML/.

Ray, E.T., 2001, *Learning XML*, O'Reilly.

Schneider, P., 2004, "Multivariate Change Point Detection in Counter-Insurgency Operations," Master thesis in Operations Research, Naval Postgraduate School, Monterey, CA (completion date September 2004).

van der Vlist, E., 2002, *XML Schema*, O'Reilly.

World Wide Web Consortium (W3C), http://www.w3.org.