Visual XML Schemas Based on Reserved Graph Grammars

Guanglei SONG Kang ZHANG
Department of Computer Science, University of Texas at Dallas
Richardson, Texas 75083-0688 USA
{gxs017800, kzhang}@utdallas.edu

Abstract:

XML (Extensible Markup Language) is becoming more and more influential in data description and information interchange. More people including none computer specialists are involved in the process of defining and using XML. As a result, it is becoming urgent to develop a user-friendly tool for describing the XML structure, which is easy and straightforward for novice to use. This paper presents a Visual XML Schema based on a graph grammar formalism. The visual approach is intuitive in describing the syntax and semantics of an XML document, and provides a visual framework for users to edit and validate XML Schema visually. This paper also presents a parsing algorithm for the visual schema, whose time complexity is polynomial.

1. Introduction

XML (Extensible Markup Language) is the base of a suite of widely used standards and recommendations for structuring, representing, exchanging and elaborating information [10]. The beauty of the XML comes from the neat definition of XML structure and separation of data and presentation. XSL (Extensible Stylesheet Language) takes charge the transformation and representation of XML, and XSD (XML Schema Definition) [5] and DTD (Document Type Definition) administer the definition of structures and data types in XML. This paper will focus on visualization of XSD, which has more advanced features than DTD. XML Schemas express shared vocabularies and provide a means for defining the structure, content and semantics of XML documents [22].

Textual languages, such as the XML Schema, are not easy for novices to learn and understand. To define data types and structures of XML documents by a textural language is difficult, tedious and error-prone. Though some editors visualize the XML schema, none of them is equipped with a formal syntactical tool to verify and parse the schema. Those editors present a set of informal graphical notations to visualize the XML Schema, but the underlying languages are still textual. The parser cannot manipulate graphical notations directly, and users have little control over graphical notations neither. Without theoretic foundation, those notations are usually defined by natural languages for users to understand. Users have to grasp the meaning of each graphical notation based on personal understanding and experience. Therefore, informal graphical notations are imprecise and ambiguous. In addition, they are not suitable for automated analysis and transformation. Using a high dimensional space, visual programming languages (VPLs) can convey complex models and concepts more expressively and intuitively than textual languages and informal graphical notations. Tree-structured documents, such as an XML document, would be concrete and intuitive to novices if expressed in a VPL.

As the underlying theory of VPLs, graph grammars provide a sound and well-established foundation in defining logical relations among the language components [17]. A graph grammar consists of a set of rules, which illustrates the way of constructing a complete graph from a variety of nodes. Graph grammars specify all legal inter-connections between individual components, i.e. any link in a valid graph can be eventually derived from a sequence of applications of grammar rules. Conversely, an un-expected link would turn out to be a violation of the graph grammar. Therefore the links denote the structure of the graph. The recently developed Reserved Graph Grammar (RGG) formalism is powerful in expressing various types of diagrams, with a parsing complexity of polynomial time under a non-ambiguous condition [23]. Based on the RGG this paper presents a graph grammar formalism, called Visual XML Schema (VXS), for defining the structure of XML documents, and any other tree-structured documents. Further the VXS parser can perform syntax-directed computations to enable dynamic XML definition, which are not addressed by current schemas.

The rest of the paper is organized as the following: Section 2 introduces the RGG and the need for adaptation for defining the tree structure of XML documents. Section 3 provides an overview of the VXS and a framework based on the VXS. Section 4 describes declarations of the VXS. Section 5 presents a parsing algorithm, and analyzes the time complexity of the

algorithm. Section 6 presents a more complex example on how the VXS works, followed by Section 7 that compares the VXS with recent related works. Finally Section 8 concludes the paper.

2. Reserved Graph Grammars and Need for Adaptation

The RGG formalism is expressed in terms of diagrams in a node-edge format, which are very similar to the "box and line" drawings [1] but devised to suit automatic analysis based on graph grammars. In a RGG, a node is organized into a two-level hierarchy as illustrated in Figure 2. A large rectangle is the first level called a *super-vertex* with embedded small rectangles as the second level called *vertices*. In a node, each vertex is uniquely identified. For convenience and simplicity, the RGG assigns a capital letter to each vertex according to the designer's convention. The name of a super-vertex distinguishes the type of nodes, similar to the type of variables in conventional programming languages. Edges are used to denote relationships between nodes. Either a vertex or a super-vertex can be the connecting point of an edge. In addition to the structural information, the RGG provides a means of associating data to nodes in terms of *attributes*. An attribute expresses a piece of data related to the component represented by a node, and can be retrieved and evaluated in the process of parsing.

A RGG consists of a set of graph rewriting rules, also called *productions*, each having two graphs that are called *left graph* and *right graph* as shown in Figure 2. A production can be applied to a given visual program (called *host graph*) in the form of an L-application or R-application. A sub-graph in the host graph is called a *redex* if it is isomorphic to the left graph in an L-application or to the right graph in an R-application. An L-application (R-application) to a host graph is to find in the host graph a redex of the left graph (right graph) of the production and replace the redex with the right graph (left graph) of the production.

Zhang *et al.* [25] present a visual approach to graphical transformations by using RGG rules to query and restructure the XML data to different formats. The approach attested that the RGG is capable of supporting automatic transformation between different XML dialects. The current RGG is however not ideal for defining the tree structure. On one hand, the RGG is designed to define diagrams that could be far more complex than trees, and need to be pruned corresponding to the requirements of tree structure definition. On the other hand, the RGG is not sufficiently expressive for defining complex tree structures, and need to be extended. Consider the simple XML Schema in Figure 1.

```
<xsd:element name="classRoster" >
   <xsd:complexType>
      < xsd:element name="student", minOccurs="1", maxOccurs="10">
         <xsd:sequence>
            < xsd:element name ="name"/>
            < xsd:element name ="id"/>
            < xsd:element name ="grade"/>
         </xsd:sequence>
      </xsd:element>
   </xsd:complexType>
</xsd:element>
```

Figure 1 ClassRoster XML Schema

classRoster may include 1 to 10 students. In order to specify the minimum and maximum number of occurrences of child elements, the RGG needs to list all the possibilities from 1 to 10. Each rule in the RGG declares a possible occurrence. It is clearly unacceptable when defining a large range of occurrences, say 1000. The RGG also has some other limitations, such as *content model*, which denotes the composition type of child elements in an XML document. To declare a content model, for example xsd:all, of an XML document the RGG needs to list all the possibilities of the content composition. If a complex data type has 10 child elements, to define the xsd:all content model of the children appearing in any order, one has to list 10!, i.e. 3628800, possible compositions.

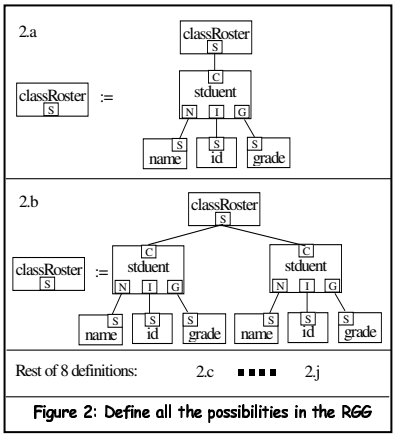

Figure 2: Define all the possibilities in the RGG

To overcome the limitations we adapt the RGG with extensions and simplifications, the modifications lead to an adapted graph grammar formalism, the Visual XML Schema (VXS). Figure 3 shows two VXS rules representing the textual XML Schema in Figure 1. The VXS defines the element student with a cardinality constraint [1..10] beside the vertex S. The number of rules is reduced from 10 in the RGG to 2, in addition the VXS is more straightforward for a novice to comprehend than the RGG in Figure 2. Intuitively the two VXS rules can be explained in natural language as following: element classRoster may include 1 to 10 students, which wraps a sequence of elements: name, id and grade. Also the content model can be addressed by the VXS and to be discussed in Section 4.2.

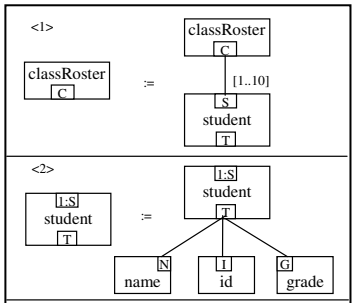

Figure 3: ClassRoster declaration in VXS

The left graph of a VXS can be simplified without causing ambiguity. In addition, the VXS simplifies the syntactical definition. A node in a tree links only to two types of nodes, parent and children, each needing only one vertex for its connection. Therefore each node needs at most two vertices. Node student in Figure 3 has only two vertices rather than four in Figure 2. The VXS differentiates children and parents by two different vertex types, thus improves the expressiveness. These modifications simplify the VXS syntax and lead to a faster parsing algorithm.

3. VXS Overview

The VXS uses node-edge diagrams to define the tree structure of an XML document. A labeled *node* (e.g. student in Figure 4) in a VXS declares a data type. An *edge* defines the relationship between a parent and a child. The position of siblings specifies the order of the children under a parent node. A production may carry a piece of *action code* [24] that customizes the syntax-directed computation. While parsing a VXS host graph, the VXS parser activates the action code associated with the production. This feature enables dynamic structure validation, which will be discussed in Section 4.

Figure 4 shows a VXS example. To define the occurrence constraints of a node, we introduce a notation, called *cardinality* (square brackets "[0..*]" of node degree), beside the *node* (degree), which defines the upper and lower bounds of node's occurrences. Cardinality will be discussed in Section 4.1. Like in the RGG, the node is organized into a two-level hierarchy, with a *super-vertex* and embedded *vertices*. The number of vertices embedded in each super-vertex in a VXS is up to 2. The vertex on top (S in student in Figure 4) links to the parent node, and the vertex at the bottom (T in student) links to child nodes. To specify the content model, we introduce a notation, , called *compositor* (the notation above vertex N of ID and D of SSN), denoted by two small diamonds connected by a dotted line covering all the underneath nodes The VXS imposes two rules on the compositor: it links only siblings; via only top-vertices. Section 4 will discuss compositors in more details.

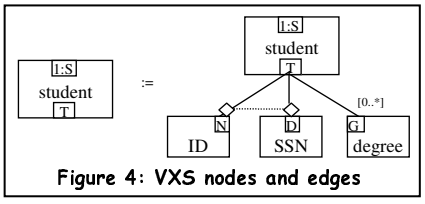

Figure 4: VXS nodes and edges

A VXS consists of a set of productions. As Figure 4 shows, every production is of the form "L:=R", with L and R being left and right graphs. When parsing a given host graph representing an XML document, the VXS parser searches for a redex of the right graph in the host graph, and replaces the redex with the left graph. This process continues until no more matches. If the graph is reduced to an empty graph, called λ, the host graph is valid. To describe the tree structure the VXS only needs one node on the left hand side and the node must be the root of right graph. Hence the VXS is a context-free graph grammar formalism, which implies that a VXS should be more efficient in parsing and require simpler specifications than its context-sensitive counterpart, the RGG.

Definition 1: A VXS graph grammar *g* is a tuple (λ, *P, T, N*), where λ is an initial graph, *P* is a set of graph grammar productions, *T* is a set of terminal labels, *N* is a set of non-terminal labels. For $\forall p = (L, R) \in P$:

1. *R* is non-empty;
2. $L \in T \cup N$, and *L* is the root of *R*;

Figure 5 shows a framework based on the VXS. A set of VXS rules specifies the structure of an XML document. A host graph conforming to a set of VXS rules visualizes the corresponding XML document. A host graph and a XML document can be converted easily to each other. The system can output the XML document based on a VXS host graph. The user specifies VXS rules and a host graph using two modules, i.e. the VXS rule generator and the VXS editor. The framework provides three major functions:

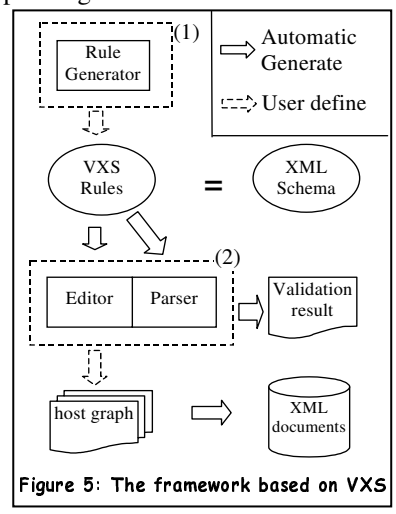

Figure 5: The framework based on VXS

1. VXS rule generator is used to specify a set of VXS rules, which defines the structure of an XML document. It compiles and integrates those rules, and automatically generates a visual programming environment，including a graph editor and a parser.
2. Based on the user supplied rules, the graph editor provides users guidance on drawing a host graph, representing an XML document in the VXS format, and prompts errors whenever the syntax is violated.
3. The parser performs syntax-directed computations. It could be used to automatically animate the validation and document-generation process and enable the dynamic structure validation.

The core part of the system is the VXS rules, based on which the editor and parser are automatically generated and thereafter perform host graph drawing, validation, and syntax-directed computation.

4. VXS Declarations

Visualization aims at illustrating the information clearly and friendly and is particularly effective in conveying structural and high level abstract information. We present the VXS based on the primitives used to define the structure of XML documents and leave the rest to the action code. As all the primitives used in most known formalisms for expressing Web data, including the XML Schema, fall into a rather limited set of categories [12] (e.g. base types, ordered sequence, un-ordered sequence, choice, cardinalities and some others) [20], we implement the following eight representatives.

1. *Base type*: the simple data type element, denoted by node attributes in the VXS.
2. *Object type:* the complex data type element.
3. *Ordered sequence, unordered sequence,* and *choice*: defined by the content model.
4. *Cardinality*: occurrence constraints of an element.
5. *Key* and *foreign key*: expressing uniqueness and referential constraints based on the content model.

4.1 Base notations

In an XML Schema, an element is declared by xsd:element. An element declaration associates a name with a data type, which is either simple or complex. The VXS represents an element of the simple type by a labeled node. Figure 6 declares two VXS simple type elements. A node in a VXS has the same name as the corresponding element in an XML Schema, Street and Zip in this case. Normally a node only shows the name and structure, because most of the time we care the structure of a document rather than its details. The VXS displays nodes in two modes, simple mode displaying only the structural information, and detailed mode showing all the detailed information.

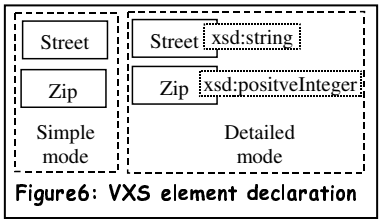

Figure6: VXS element declaration

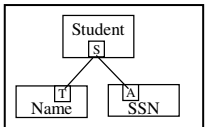

Figure 7: VXS structure declaration

The VXS declares each element of a complex data type as a parent node linked to its children. Figure 7 defines three elements, Student, Name and SSN, connected by two edges. The order of siblings under the same parent is determined by the left-right order, e.g. Name is the first child of Student.

4.2 Occurrence Constraints

The VXS denotes the occurrence constraints by two numbers in the cardinality of a node. The first number denotes the minimum occurrences and the second denotes the maximum occurrences. Asterisk (*) denotes any number, and plus (+) denotes a number from 1 to unbounded. If the cardinality is absent, both minimum and maximum occurrences are 1 by default. Figure 8 illustrates an example of VXS occurrence constraints, where Book has a cardinality of "[0..*]", indicating 0 to unbounded number of occurrences. A cardinality "[1..5]" associated with vertex F of node Author specifies that Author has 1 to 5 possible occurrences.

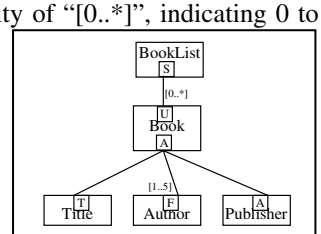

Figure 8: Occurrence constraints

4.3 Content Model

When an element declaration or a wildcard is used in a complex type definition to constrain the valid children of an element, such a declaration is called a *content model*. The XML Schema has three types of content models defined by three compositor elements: xsd:sequence, xsd:choice, and xsd:all. Table 1 shows the description of these three types.

Table 1: Description of Compositors

Compositors Name	Description
xsd:sequence	Elements must occur exactly in the order indicated and all must occur (unless declared by minOccurs otherwise).
xsd:choice	Exactly one of the grouped elements may occur, and the elements in the group are mutually exclusive.
xsd:all	All of the grouped elements may appear, but in any order. minOccurs is either 0 or 1, and maxOccurs must be 1.

To declare a content model concisely, the VXS uses a compositor to connect vertices hosted in sibling nodes, drawn in dotted line.

All the three types of compositor elements are shown in Figure 9, with various types of compositor elements: a dotted line connects vertices of sibling nodes as an indicator of special content model; diamonds in the ends of compositor-edge determine the type of content model either xsd:choice or xsd:all. If a compositor is absent, the content model is xsd:sequence by default. Otherwise if nodes are linked by a dotted line with empty diamonds in both ends, the content model is xsd:choice. Solid diamond denotes xsd:all. In Figure 9, Student has a sequence of child elements, which is in the same order as in the XML Schema and the VXS shows the sequence in the default mode with no extra edge. In Figure 9 a compositor links two elements, StudentID and SSN, by a dotted line with empty diamonds in both ends and

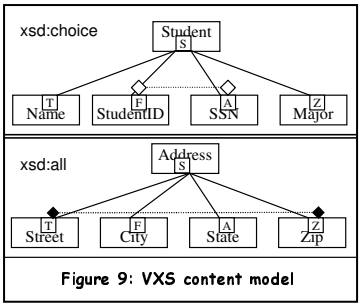

Figure 9: VXS content model

denotes that the two elements are in the xsd:choice model. Also Figure 9 defines xsd:all of all the child nodes of Address.

4.5 Custom Codes

The VXS inherits RGG's ability to embed action-code. In addition, the VXS adds two types of custom codes to each node: *content-code* and *attribute-code*. An *Attribute-code* represents attributes of associated node, and a *content-code* declares elements enclosed by associated node. The content-code is essentially a supplementary textual language, which is able to declare any structure and data type in an XML Schema. The VXS can also declare a key or a foreign key by using the content-code.

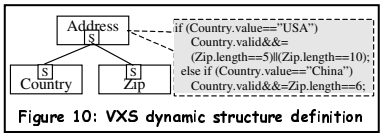

Figure 10: VXS dynamic structure definition

With action-code the VXS parser is able to perform syntax-directed computations, which enable dynamic structure definition. For example, element Address is a complex data type, which encloses Country and Zip. In different countries the format of Zip code may be different, for example, 6 digits in China, 5 or 10 digits in USA. With the action-code the VXS can define the Zip pattern related to the value Country. The pseudo action-code of Address could be written as in Figure 10. When parsing a host graph, the parser matches the production against the host graph, and applies the production. The parser meanwhile performs the action-code, which checks the value of the Zip according to the value of Country, and then reports invalidation if the value does not match. Therefore the VXS can define dynamic structures of XML documents.

4.6 Functional Equivalence

It is easy to prove that the VXS is functionally equivalent to the XML Schema language. Here "functionally equivalent" means that the VXS can describe any XML structure defined by the XML Schema language.

An XML Schema defines the structure of any XML document. In a VXS, each node denotes an element declaration, and the edge denotes a parent-child relationship, which represents the structural relationship between elements. The parent-child relationship is the only relationship that determines the overall structure of a tree. Therefore the VXS can completely describe any tree structure of XML documents defined by an XML Schema.

An XML Schema also defines some non-structured information, such as attributes, which a node-edge diagram cannot define. But the embedded VXS custom codes can describe such information inherently using the supplementary textual language. Therefore the VXS is able to declare any XML Schema, i.e. the VXS is functionally equivalent to the XML Schema language.

5. Parsing the VXS

This section presents a parsing algorithm for the VXS.

5.1 A Parsing Algorithm

The VXS parsing algorithm is different from that of the RGG, because in the VXS matching a right graph with a host graph is a tree-to-tree match rather than graph-to-graph match in the RGG. The tree-to-tree match can be performed much faster when using appropriate data structure and algorithm. Figure 11 illustrates the pseudo code of a parsing algorithm.

```
Parsing (HostGraph host)
{
    while host  graph is not empty do
    {
        matched = false;
        for all p∈ Productions
        {
            Redex = FindRedex (host, p);
            If (Redex is not empty)
            {
                ApplyRedex(host, p, Redex);
                Matched = true;
            }
        }
        if (matched == false)
        {
            print ("Invalid");
            exit(0);
        }
    }
}
FindRedex (HostGraph host, Production p)
{
    For every node in host graph
    {
        if the node matches the root of production p
        {
            match p as the sub tree of the node;
            if not match, continue;
            return the node and any node matches in production;
        }
    }
    return null;
}
ApplyRedex (HostGraph host, Production p, Redex r)
{
    Find the root of Redex's Right graph;
    Delete redex in host graph, and reserve the root.
}
```

Figure 11 VXS parsing algorithm

In the VXS the parsing process is a sequence of R-applications, which is modeled as recognize-select-execute [3]. The parsing process proceeds as following:

1. Search for a redex of the right graph in the host graph, i.e. function FindRedex in Figure 11;
2. Embed a copy of the left graph (i.e. the root of the right graph) into the host graph by replacing the redex, i.e. function ApplyRedex.

3. Repeat Steps 1 and 2 until the host graph is empty or no more matches for any productions.

Two or more occurrences of a right graph may exist in a host graph during, and the application order of the productions may affect the parsing result. Even for the most restricted classes of graph grammars, the membership problem is NP-hard [16]. As a consequence, a parser may not recognize a syntactically correct graph or be inefficient in analyzing a large and complex graph. To allow efficient parsing without backtracking, we are only interested in *confluent* graph grammars. Informally, the confluence means that different orders of applications will result in the same result. The parsing algorithm in Figure 11 regards the VXS as a confluent grammar, which only tries one parsing path. The following sub-section will prove that any VXS grammar is confluent.

5.2 Analysis

We now prove that in the VXS any order of applications of productions will lead to the same result, i.e. the VXS is a confluent graph grammar.

Theorem 1. Any VXS grammar is a confluent graph grammar.

Proof:

Hypothesis: suppose there are two different productions, **R1** and **R2**, which lead to different results when applied in different orders.

1. If $R1 \cap R2 = \phi$, i.e. R1 and R2 have no common node, they will lead to the same result in any application order, because after applying each production to any redex, the redex becomes a leaf node of the intermediate tree and the application does not change other redexes. Part [1] of Figure 12 shows that the parser parses R1 and R2 to leaf nodes and obtains the same intermediate tree after both are applied since the applications of R1 and R2 are independent to each other. Hence, $R1 \cap R2 \neq \phi$, i.e. R1 and R2 must have at least one common node to lead to different results in different application orders.

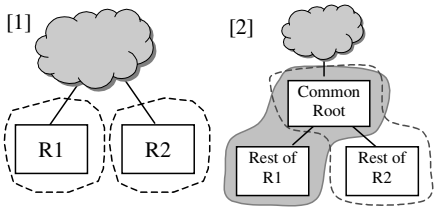

Figure 12 Illustration of the proof

2. Suppose $R1 \cap R2 = \{root\ (R1)\} = \{root\ (R2)\}$, i.e. the root is the only node R1 and R2 have in common. As part [2] of Figure 12 shows, the application of R1 (redex in the shadowed area) or R2 (redex in a dashed line) will remove all the nodes except the root, together with which the intermediate tree can have a match with the other production. Then the second production application will produce an intermediate tree, which includes the common root of R1 and R2. In any order, therefore, the applications of the two productions will lead to the same result. Hence the assumption is false, i.e. R1 and R2 must have at least one more node in common apart from the root, e.g. $R1 \cap R2 > \{root\ (R1)\} = \{root\ (R2)\}$.

Thus the parser will not generate a valid result no matter what the application order is if R1 and R2 are different productions. Suppose node x belongs to $R1 \cap R2$, and x is not the root. If one production applies first, the parser will remove the redex except the root from the host graph, e.g. the x is removed, and another production will not match without x. Thus the result of the parsing will always be invalid, i.e. lead to the same result. Therefore R1 and R2 cannot be different productions. That is a contradiction to the hypothesis.

Based on Steps 1 and 2, we draw the conclusion that the applications of the R1 and R2 always lead to the same result. Hence the theorem is proved. □

Theorem 2. The time complexity of the VXS parsing algorithm is O ($M*N^3$).

Proof:

Suppose we have a VXS host graph, with **N** nodes. A right graph has at most **M** nodes.

The parsing algorithm includes two loops, and in the worst case each loop will iterate N times. That is O (N^2) running time. The procedure FindRedex also has two loops, and the outer loop searches every node in the host graph, which requires N running time. The other loop checks if a production matches. Matching the right graph with the host graph, both being trees, is done by traversing the right graph. Therefore the match procedure requires M operations. The FindRedex requires O (M*N) running time. To embed the left graph, ApplyRedex requires M running time.

Hence the overall parsing time is O (N^{2*}(M*N+M)), or O ($M*N^3$). □

6. A More Complex Example

This section will go over a more complex example to give an overall idea on how the VXS may be used. Figure 13 lists an XML Schema fragment that declares a complex type called classRoster, which includes 1 to 10 students. The student is also a complex type, and so on. For a novice, the XML Schema is not straightforward to comprehend.

```
<xsd:element name="classRoster" >
   <xsd:complexType>
      < xsd:ref name="student", minOccurs="1", maxOccurs="10">
   </xsd:complexType>
</xsd:element>

<xsd:element name="Student">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="Name" type="xsd:string"/>
            <xsd:choice>
               <xsd:element name="ID", type="xsd:string"/>
```

```
                <xsd:element name="SSN", type="xsd:string"/>
            </xsd:choice>
            <xsd:element ref="Address"/>
            <xsd:element name="Major" type="xsd:string", maxOccurs=
"unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="Address">
    <xsd:complexType>
        <xsd:all>
            <xsd:element name="Street" type="xsd:string"/>
            <xsd:element name="City" type="xsd:string"/>
            <xsd:element name="State" type="xsd:string"/>
            <xsd:element name="Zip" type="xsd:positiveInteger"/>
        </xsd:all>
    </xsd:complexType>
</xsd:element>
```

Figure 13: A complex XML Schema

Figure 14 shows a corresponding VXS, which explicitly and precisely declares the XML Schema. Comparing to the textual schema in Figure 13, the VXS is more intuitive and concrete. A host graph based on the VXS is shown in Figure 15 with parsing steps annotated in the dashed enclosures.

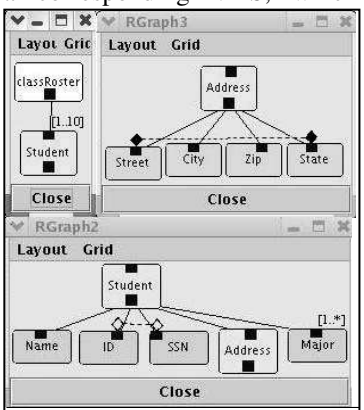

Figure 14: The classRoster schema in VXS

The XML document has a root classRoster, which includes two students. Each student has the same sequence of child elements Name, ID, Address, and etc. To validate the XML document against the schema, the VXS parser parses the host graph against the VXS rules defined in Figure 14. Each part of the tree encircled by a

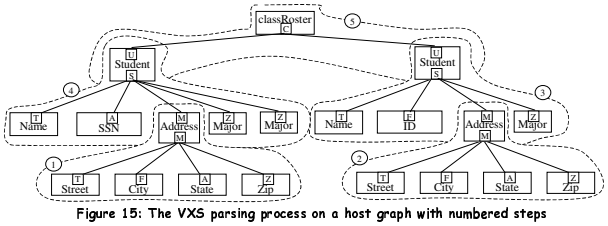

Figure 15: The VXS parsing process on a host graph with numbered steps

dashed line is a redex matching the right graph. After the parser finds a redex in the tree, it will replace the redex with the left graph of the rule, i.e. the root of the right graph. In this example, the parser matches Address element first, and all the child elements of Address are removed after the R-application of rule <3>. The parser the process, until no match in the host graph can be found. If at the end of the parsing process the host graph is reduced to λ, the host graph is valid, otherwise it is not. In Figure 16 the host graph is reduced to λ after 5 steps of R-application, therefore the host graph is valid, i.e. the corresponding XML document is valid against the VXS grammar, i.e. the XML Schema. Figure 16 shows the host graph in our system.

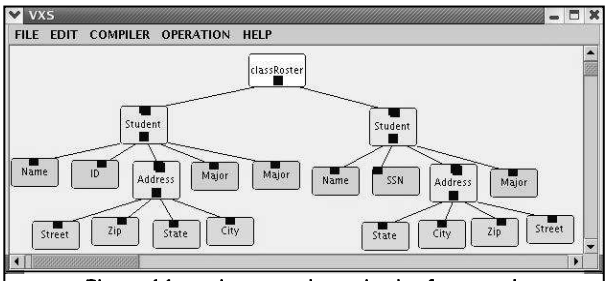

Figure 16: an instance drawn in the framework

7. Related Work

Several XML visualizations have been proposed or implemented, such as XML-GL [6], and Xing [13]. XML-GL is a visual language for querying and restructuring XML data by node-edge diagrams. It achieves XML transformation and restructuring using some rules that combine the patterns of queries and results returned by queries. Those languages commonly utilize highly sophisticated mechanisms, which are too complex for ordinary users. Also some XML Schema editors are visualized, such as VXT [9], SchemaViewer [19]. SchemaViewer shares similar ideas with our approach, but it is weak in the control over visual objects without a theoretic foundation.

On the other hand several alternatives to the XML Schema have been proposed, such as RELAX [15], TREX [21], and Schematron [18]. Makoto proposed RELAX (REgular LAnguage description of XML), which is based on hedge automata theory and generated considerable interest as a simpler solution than XML Schema. RELAX permits the use of XML syntax, borrows datatypes from XML Schema Part 2, and supports namespaces. In contrast to the grammar-based approaches, Jelliffe's Schematron is fundamentally different in that it is based on pattern matching using the Xpath [7] expressions used by XSLT [8]. Alternatives to XML Schema have gained popularity. Using texts, they are not as intuitive as graphs in the VXS. The languages are not formalized. Brown etc. present MSL (Model Schema Language) [2] to formalize some of the core ideas in XML Schema and proved the necessary of formalization. MSL has already proved helpful in the work on the design of XML Query. Based on graph grammars, our approach is an inborn formalized language.

Bernstein [4] specifies a more wide range of operators for heterogeneous schemas, including XML Schemas and SQL schemas. A fundamental operation in the manipulation of schema information is match.

According to the taxonomy of the schema matching [14], our approach falls into a category of graph matching.

8. Conclusion

This paper has proposed a visual formalism, the VXS, for specifying XML Schemas, and provided a framework based on the VXS for users to construct and validate schemas visually. As a graph grammar, the VXS addresses the fundamental parsing problem, i.e. matching and embedding of right graph. As stated in theorem 1, the VXS is always a confluent grammar, thus the parsing procedure can be performed in any order when the host graph has multiple redexes. Parsing will generate the same result regardless the order of applications. This paper presented a parsing algorithm with polynomial time complexity. The parsing algorithm is simplified from that of the RGG and is faster.

Comparing to other schemas, the VXS has the following advantages: as a visualized schema, it is simpler and easier for novice to use than textual languages; as a context-free graph grammar, a VXS is precise and succinct, yet efficient in parsing.

References

[1] R. Allen and D. Garlan, Formalizing Architectural Connection, *Proc. 16th International Conference on Software Engineering*, 1994, pp.71-80.

[2] Allen Brown, Matthew Fuchs, Jonathan Robie, Philip Wadler, MSL: a model for W3C XML Schema, *Computer Networks 39*, 2002, pp. 507 - 521.

[3] R. Bardohl, G. Taentzer, M. Minas, and A. Schürr, Application of Graph Transformation to Visual Languages, *World Scientific*, 1999, pp.105-180.

[4] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems, *Proc. of the 2003 CIDR Conference*, 2003.

[5] P. V. Biron and A. Malhotra (ed.). XML Schema Part2: Datatypes. *W3C Document*, April 2000. http://www.w3.org/.

[6] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tancia, "XML-GL: a graphical language for querying and representing XML documents", *Proc. of 8th International Conference on World Wide Web,* 1999.

[7] J. Clark, S. DeRose (ed.) XML Path Language (XPath) Version 1.0, *W3C*, *Internet Document*, November 1999. (http://www.w3.org/TR/xpath).

[8] J. Clark (ed.), XML Transformations (XSLT) Version 1.0, *W3C*, November 1999. (http://www.w3.org/TR/xslt).

[9] Pietriga, E., Quint, V., Vion-Dury, J.Y, VXT: A Visual Approach to XML Transformations. *ACM Symp. on Document Engineering*. 2001.

[10] E.R. Harold, *The XML Bible*, 2nd Edition, Hungry Minds, 2001.

[11] Kenneth B. Sall, *XML Family of Specifications, A Practical Guide*, Addison-Wesley, 2002, pp. 253 – 373.

[12] D. Lee and W. W. Chu. Comparative Analysis of Six XML Schema Languages. *SIGMOD Record*, 29(3): 2000, pp, 76-87.

[13] Erwig, M.: A Visual Language for XML. *IEEE Symp. on Visual Languages*, 2000. pp. 47—54.

[14] E. Rahm, P. A. Bernstein, A survey of approaches to automatic schema matching, *The VLDB journal* 10, 2001: pp. 334-350.

[15] M. Makoto, RELAX (REgular LAnguage description for XML), *Internet Document*, April 2000. (http://www.xml.gr.jp/relax/).

[16] G. Rozenberg and E. Welzl, Boundary NLC graph grammars – basic definitions, normal forms, and complexity, *Information and Control*, Vol.*69*, 1986, pp.136-167.

[17] G. Rozenberg (ed.), Handbook on Graph Grammars and Computing by Graph Transformation: Foundations, *Vol.1, World Scientific*, 1997.

[18] R. Jelliffe, Schematron, *Internet Document*, May 2000. (http://www.ascc.net/xml/resource/schematron/).

[19] F. Kilkelly, SchemaViewer1.0, *Internet Document*, Nov. 2002. http://xml.coverpages.org/SchemaViewer10-Ann.html.

[20] P. Atzeni, R. Torlone. A Unified Framework for Data Translation over the Web. *2nd International Conference on Web Information System Engineering (WISE 2001)*, Kyoto, Japan, 2001.

[21] James Clark, TREX - Tree Regular Expressions for XML, *Internet Document*, Jan 2001. http://www.thaiopensource.com/trex/

[22] D.C. Fallside (Ed.), XML Schema Part 0: Primer. *World Wide Web Consortium*, Recommendation, 2 May 2001.

[23] D. Q. Zhang, *Generation of Visual Programming Languages*, Ph.D. Thesis, Macquarie Univ., 1998.

[24] K. Zhang, Design, Construction, and Application of a Generic Visual Language Generation Environment, *IEEE Transactions on software engineering*, vol. 27, no. 4, April 2001.

[25] K. Zhang, Graphical Transformation of Multimedia XML Documents, *Annals of Software Engineering* 12, 2001, pp. 119-137.