Computers & Geosciences 31 (2005) 1135–1150

www.elsevier.com/locate/cageo

# Modeling geological objects with the XML Schema

Hassan A. Babaie[a,*], Abbed Babaei[b]

[a]*Department of Geology, Georgia State University, Atlanta, GA 30303, USA*
[b]*Department of Biol., Geol. & Env. Sci., Cleveland State University, Cleveland, OH 44115-2214, USA*

Received 14 May 2004; received in revised form 6 October 2004; accepted 31 December 2004

**Abstract**

Interchange, storage, and management of geological data require the development of knowledge-based, standardized vocabularies and data structures. Concepts modeled and designed with the Unified Markup Language (UML), can be mapped into XML Schema Definition Language (XSDL) to compose modular markup languages for each discipline. Developing such efficient, intra-disciplinary, modular and reusable components, based on the XSDL namespace facility and the principles of object-oriented design, reduces redundancy, increases efficiency, scalability, and extensibility, and simplifies the maintenance and future extension of the code.

This paper discusses the best practices of composition and reuse of modular intra-disciplinary components by applying XML Schema namespace syntax. In addition to several small examples given for a variety of geological cases, the paper constructs a UML conceptual model and markup language, applying an XML-type library, for a component of the plate tectonics knowledge base (TectonicsML) that deals with the divergent plate boundary.

*Keywords:* XML Schema; Knowledge management; Markup language; Modeling with UML; Metadata

## 1. Introduction

Storage, interchange, and sharing of data in geosciences are facing new challenges as a result of the Internet and the World Wide Web and emergence of new data management paradigms. The challenges are mainly due to the need for cross-disciplinary, intra-domain and cross-field, inter-domain system developments, and the lack of data standards in these fields to support integration of complex data structures.

Definition and integration of the metadata that describe the format, constraints, structures, and rules for the scientific data, are necessary for successful sharing, storage, and management of data and information in the field. Cross-disciplinary integration of data requires modification of the general data structures, which can become complicated and inefficient if mechanisms of reuse, extension, scope recognition (e.g., namespace), and restriction of the metadata do not exist.

A discipline-specific, scientific markup language reflects the syntax and structure of the concepts of the knowledge base of the discipline, and standardizes the exchange, storage, and retrieval of data in that discipline. XML Schema's

*Corresponding author. Tel.: +1 404 463 9559; fax: +1 404 651 1376.
*E-mail address:* hbabaie@gsu.edu (H.A. Babaie).

0098-3004/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cageo.2004.12.009

namespace (van der Vlist, 2002[1,2]) is of great value, giving every discipline and field a unique scope and allowing smooth integration without any conflict.

The objectives of this paper are to: (i) introduce the application of the Unified Modeling Language[3] for designing the conceptual model of a knowledge base, (ii) introduce the knowledge management and the W3C XML Schema Definition Language (XSDL);[1](iii) apply the best practices of the XML Schema language in the development of markup languages in any geological discipline; (iv) give a geological example to elaborate the modular composition of schemas. The emphasis of the design and modeling will be on schema composition with cross-discipline and cross-field reuse and modularization in mind, applying the XML Schema's reuse and namespace technologies.

## 2. Background

Whereas data are discrete and self-contained bits and numbers (e.g., N45°E, 10 °C, porphyritic, grain, 124 m), with no meaning of their own, information is acquired by arranging and organizing a selected, filtered set of data, in a specific context, so that they have meaning and could be communicated with other people (e.g., Rumizen, 2002). For example, the drawdown data in an unconfined aquifer are just numbers representing the depth to the water table at different times. Only when selected and plotted (i.e., arranged) on specific diagrams (e.g., Theis curve), the numbers become meaningful and useful (e.g., reveal single- or dual-porosity medium), from which other information (e.g., hydraulic conductivity) could be extracted, or an action (e.g., remediation) could be taken. The value of the information is intrinsic to the information user (Gillette, 2002), i.e., the drawdown data may not be useful to a non-hydrogeologist. This means that information to a hydrogeologist may be data to a chemist, and vice versa.

Information, obtained from many sources (e.g., experience, experiment, analysis, test), represents how knowledge moves between things and people (Gillette, 2002). New information leads to a new knowledge, which creates more information, which deepens the knowledge (Watson, 2003). Knowledge involves activities such as creating, identifying, sharing, capturing, acquiring, and leveraging, and is constantly changing (Rumizen, 2002). Explicit knowledge includes things that can be written down, codified, shared with other people, or put in a database (e.g., Rumizen, 2002). Tacit or implicit knowledge represents things that we do not know that we know; it can be felt and understood but not expressed, and includes intuition, insight, experience, rules of thumb, and know-how.

Information forms when we get an understanding of the relations among the data, and knowledge forms when we recognize and understand the meaning of the patterns in a given context. For example, given data about the attitude of foliation and fold elements (isolated numbers—data) in deformed slate (context), a stereographic projection of the data (selecting and plotting—information) will reveal parallelism of the foliations and isoclinal fold axial planes and axes (revealing a pattern—information), which may mean that the fold and foliation formed at the same time (understanding the pattern—knowledge). In other words, the pattern (i.e., parallelism of the axial plane and foliation) represents a pre-existing knowledge. Based on this knowledge, we then take an action, such as find out if other contractional structures (e.g., thrust faults) exist in the area, calculate the amount of shortening, or decide to undo the shortening of the slate, to find the pre-deformation state of the rock.

According to the principle of diminishing information (Kähre, 2002), transmission of information through intermediary channels leads to a decrease in the amount of information when transmission is completed. One way to reduce the rate of diminishing information during data transmission is to standardize the structure, storage, and transmission of data by developing markup languages and databases. A domain-specific markup language attaches context to the value of data and allows data to transmit information (meaning), making it more useful and less prone to loss or corruption.

## 3. Designing a markup language

### 3.1. Conceptual modeling

Knowledge management includes all the continuous ways in which the knowledge in a specific field or domain (e.g., sedimentology) is acquired (i.e., learned, created, identified), interchanged (e.g., on the Web), stored (e.g., in a

[1]XSDL, 2001. XML Schema: Part 0: Primer, http://www.w3.org/TR/xmlschema-0; Part 1: Structures, http://www.w3.org/TR/xmlschema-1.

[2]Namespace Specification, http://www.w3.org/TR/REC-xml-names.

[3]UML, 1977. Unified Modeling language, http://www.omg.org/technology/uml.

database), applied, and updated (Watson, 2003). It is obvious that we use knowledge to acquire new knowledge through information. Thus, knowledge management is essential for any field or sub-discipline for recognizing the underlying assets, and knowing what the members of the field know and how they know and apply it. One goal of knowledge representation and management in geology is to make the explicit and implicit geological knowledge accessible to humans and knowledge-based and object-oriented computer applications, through high-level languages (e.g., XML[4]) that are close to human languages.

A conceptual model represents a modeler's perception of the field or domain (e.g., geology). When designed with Unified Markup Language (UML), (see footnote 3) the conceptual model becomes a starting point from which other implementations can be derived. The first step in designing a markup language, database, object-oriented system, or knowledge base, to represent a discipline, is to define the vocabulary for the domain (i.e., field) concepts. These concepts (e.g., aquifer, fault, earthquake) will be represented by 'classes' in object-oriented languages such as Java and UML.

It is important to think ahead of time for the future extensibility (i.e., evolution) of the markup language. As the science in the domain grows with time, the model should accommodate these changes in the best and least disruptive ways. To ensure an iterative and incremental development (Larman, 2002; Eeles et al., 2002), we must select (i.e., abstract) a subset of the concepts in the field by deliberately ignoring minor aspects of the system in earlier versions. In order to comprehend and handle all aspects of this subset of the system, we must decompose it into several subsystems or packages, a task which is best done with UML.

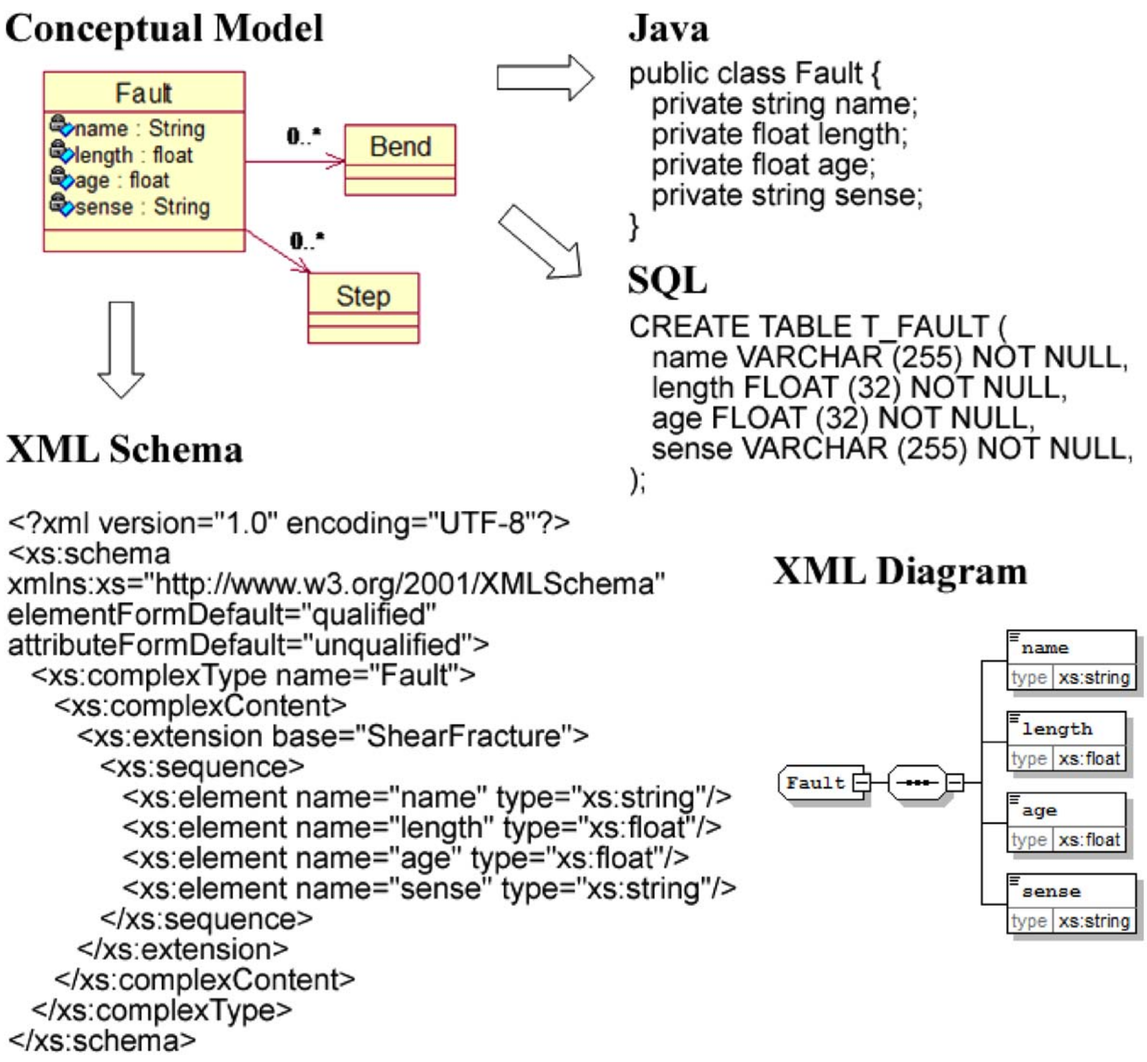
### *3.1.1. Unified Modeling Language, UML*

UML is used to specify a markup language by building precise, unambiguous, and complete models of the system. UML leads to a better understanding of the system by providing means of viewing the system from different perspectives (e.g., structural, behavioral, functional). For example, UML class diagrams define the structure of geological concepts such as aquifer, well, and porosity in the hydrogeology domain. UML is made of three kinds of building blocks: things, relationships, and diagrams (Booch et al., 1999). Things can be of four kinds: structural, behavioral, grouping, and annotational. Structural things are the nouns in the domain, and represent the static, conceptual, or physical parts of the model (e.g., mineral, rock, texture, magma). The structural things, represented on the UML diagram with classes, are descriptions of a set of objects (e.g., bedding, unconformity) that share the same attributes, operations, relationship (e.g., composition, aggregation), and semantics (Booch et al., 1999). Classes are shown with a rectangle with three compartments (Figs. 1 and 5). The top compartment takes the name of the class or asset, and may have other ornaments on the UML class diagram, such as stereotype and package identification. When translated into XML Schema, classes become elements with a complex type, and their attributes convert to sub-elements or attributes. To guarantee proper conversion of UML model to XML Schema code (e.g., Carlson, 2001), all packages and classes must be designated with the ⟨⟨XSD Schema⟩⟩ and ⟨⟨XSDcomplexType⟩⟩ stereotypes, respectively (Babaie and Babaei, 2005). The UML ⟨⟨XSDcomplexType⟩⟩ class stereotype allows mapping of the class to both attributes and elements in the XML Schema. UML association role names, with cardinality more than 1, also convert to elements.

The second compartment is reserved for attributes representing the characteristics of the objects, which are shown with the first character in lowercase (e.g., strike, thickness, age). The third compartment shows the list of the operations, representing the methods in the object-oriented programming languages. The operations represent the behavior of the objects of a class, and are defined if we want to implement the XSL Schema into Java, VB.NET, or C+ + code. The grouping things are used to organize the UML model, and represent decompositions or subsystems of the system, represented by packages (a group of classes). The annotational things are the explanatory parts of the UML diagram, and are the comments created to illuminate and describe other elements of the model. These comments are created with the note element, which is shown with a rectangle with a dog-eared corner, and contain an explanation or remark.

There are four kinds of relationships between classes in a UML model: dependency, association, generalization, and realization. Dependency represents a semantic relationship between two things in which a change in one thing affects the semantics of the dependent thing. This relationship is visually rendered as a dashed line, and may be directed from the dependent thing to the independent thing. For example, the Deformation and FaultRock packages both may depend on the RockStructure package. An association is a structural relationship that describes a set of links between objects (classes). A special kind of association is called aggregation, representing a relationship between a whole (e.g., Rock) and its parts (e.g., Mineral, Grain, Cement, Matrix). Association is shown with a solid line, uni- or bi-directional, with or without label, commonly showing cardinalities and role names. Cardinalities indicate the number of objects that participate in the relationship (e.g., 0...1 means zero to one objects are related).

[4]XML, 1998. XML 1.0 Recommendation, http://www.w3.org/TR/REC-xml.



Java

```
public class Fault {
  private string name;
  private float length;
  private float age;
  private string sense;
}
```

SQL

```
CREATE TABLE T_FAULT (
  name VARCHAR (255) NOT NULL,
  length FLOAT (32) NOT NULL,
  age FLOAT (32) NOT NULL,
  sense VARCHAR (255) NOT NULL,
);
```

XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <xs:complexType name="Fault">
    <xs:complexContent>
      <xs:extension base="ShearFracture">
        <xs:sequence>
          <xs:element name="name" type="xs:string"/>
          <xs:element name="length" type="xs:float"/>
          <xs:element name="age" type="xs:float"/>
          <xs:element name="sense" type="xs:string"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

Fig. 1. Forward engineering (mapping) of abbreviated Fault conceptual model into Java code, SQL (DDL), and XML Schema. Implementations of Step and Bend classes are not shown. 0...* cardinality, directed from Fault to Bend, means that a fault segment may have zero to many bends.

A generalization is a relationship in which the objects of the specialized element (the child, e.g., TransformFault) are substitutable for objects of the generalized element (the parent, e.g., StrikeSlipFault). In UML, generalization is depicted graphically with a solid line with a hollow arrowhead pointing to the parent. The specialized, child class (e.g., PWave) shares the structure and behavior of its more general, parent class (BodyWave) but can define additional attributes and/or operations, and can be used anywhere in an application where its parent can be used. This provides a mechanism for the powerful inheritance and polymorphic behavior for object-oriented code efficiency. Realization is a semantic relationship between a class and an interface, and is out of the scope of this paper.

### 3.2. Extensible Markup Language, XML

The XML technology is designed for structuring data through markup. It does this through tags, similar to the HTML tags, although a lot more powerful. While HTML tags only format the text, e.g., Plate Tectonics) which renders the 'Plate Tectonics' string in bold letters, XML reflects the structure of the data. For example, the snippet XML code below, defines three tags that convey some information about their meaning and structure to humans (but not to machines!). Here, we have a PlateTectonics element (Babaie and Babaei, 2003) that contains two other elements called ForeArc and BackArc reflecting part of the plate tectonics knowledge base.

```
<PlateTectonics>
   <ForeArc> Tobago </ForeArc>
   <BackArc> Caribbean </BackArc>
</PlateTectonics>
```

To make this document code more meaningful to both human and machine users, we define what is called a schema applying the W3 XML Schema Language. The schema must be accessible to the document so that an XML editor, parsing the document, can validate the document against it. Tags can be any thing we want them to be, and of any length, as long as they make sense to their targeted human users. For example, the Backarc and Geobarometer tags are more informative to a geologist than their 'Ba' or 'Gb' short versions. The data structure reflected by the markup is independent from how the data are rendered. This makes it possible to deliver information from the same source in multiple formats such as print and online (Goldfarb and Prescod, 2004).

### *3.2.1. W3C XML Schema*

Concepts in the domain knowledge base become elements and attributes in the XML Schema (Binstock et al., 2003). Elements, identified by a tag name, can have a simple or complex type. The XML Schema defines its own built-in primitive types such as xs:string, xs:float, and xs:time, that could be used to construct (i.e., derive) domain-specific types. For example, the following XML Schema defines a RefractiveIndex type for a mineral by restricting the primitive xs:decimal type and allowing only two decimal points (e.g., 2.34).

```
<xs:simpleType name="RefractiveIndex">
   <xs:restriction base="xs:decimal">
      <xs:fractionDigits value="2"/>
   </xs:restriction>
</xs:simpleType>
```

We can construct our own data types from the primitive types by restriction or extension, and applying constraints. For example, in the above example, we used restriction and constrained the primitive decimal type to two decimal points by defining the fractionDigits facet for the xs:decimal type. The following code snippets restrict the xs:integer and xs:string primitive types to define the Azimuth (e.g., 232°) and Quadrant (S52°W) types (for planar geological objects such as fault and bedding) using the xs:pattern syntax. The ⟨!-- ... --⟩ statements are XML annotations (i.e., comments).

```
<!-- Azimuth format such as 045° -->
<xs:element name="Azimuth">
   <xs:simpleType>
      <xs:restriction base="xs:integer">
         <xs:maxInclusive value="360"/>
         <!-- allow a value between 0 and 360 -->
         <xs:pattern value="[0-3][0-9][0-9]"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<!-- Quadrant format such as N45E or S42W-->
<xs:element name="Quadrant">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:length value="4"/>
         <xs:pattern value="[N|S]([0-8][0-9])[E|W]"/>
         <xs:pattern value="[N|S]([9][0])[E|W]"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
```

Geological concepts, like those in other domains, have complex structures. Elements often have a hierarchical structure with several layers of nested elements which are themselves nested in a root element. To reduce complexity of such structures, we often maximize element reuse by defining type libraries (Babaie and Babaei, 2005) and applying the

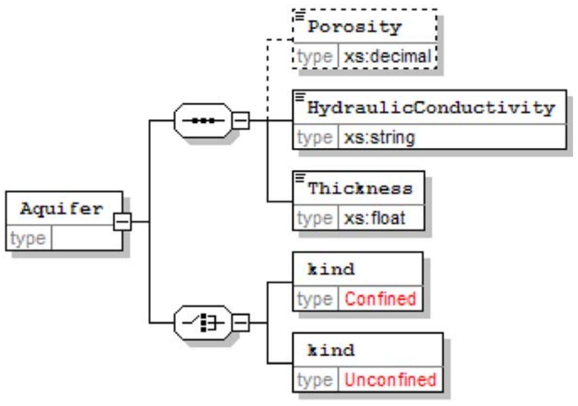


Fig. 2. XML Schema diagram for Aquifer element, using sequence and choice compositors. See Listing above for definition of Aquifer. Confined and Unconfined types are yet to be defined.

namespace mechanism. Elements have a name, and could be of simple or complex type. The complex type is used to aggregate several elements and attributes in a new data type. Only the leaf node elements, i.e., those without child elements and attributes, are simple type. We define the complex types at the global or local level. The following is the definition of a local complex hydrogeological type called Aquifer. The XML diagram of the simplified Aquifer element is given in Fig. 2. The definitions of the Confined and Unconfined elements are not given here for brevity.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="Aquifer">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Porosity" type="xs:decimal" minOccurs="0"/>
        <xs:element name="HydraulicConductivity" type="xs:string"/>
        <xs:element name="Thickness" type="xs:float"/>
      </xs:sequence>
      <xs:choice>
       <!—the Confined & Unconfined types are defined elsewhere -->
       <xs:element name="kind" type="Confined"/>
       <xs:element name="kind" type="Unconfined"/>
      </xs:choice>
      <xs:attribute name="Name" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Complex types contain one or more particles defined with the sequence, choice, or all connectors. The Aquifer complex type, defined above and depicted in Fig. 2, has two particles and an attribute. The first particle has three elements, connected with the sequence connector (compositor), and the second particle has two elements constructed with the choice connector. The Porosity element in the first particle is optional as constrained by the minOccurs facet (shown with dashed borders). The Name attribute must be provided in the XML instance which will be based on this schema, as constrained by the use = “required”. Elements and particles have cardinality constraints specified with minOccurs and maxOccurs. The sequence connector indicates that porosity, hydraulic conductivity, and thickness must be given in that order in an XML document (instance). If order is not important, then the all connector must be used instead of sequence. The choice connector provides a set of alternative elements, only one of which must be used. In the above example, the aquifer type is stated to either be confined or unconfined. These two types must be defined in the schema at the global level.

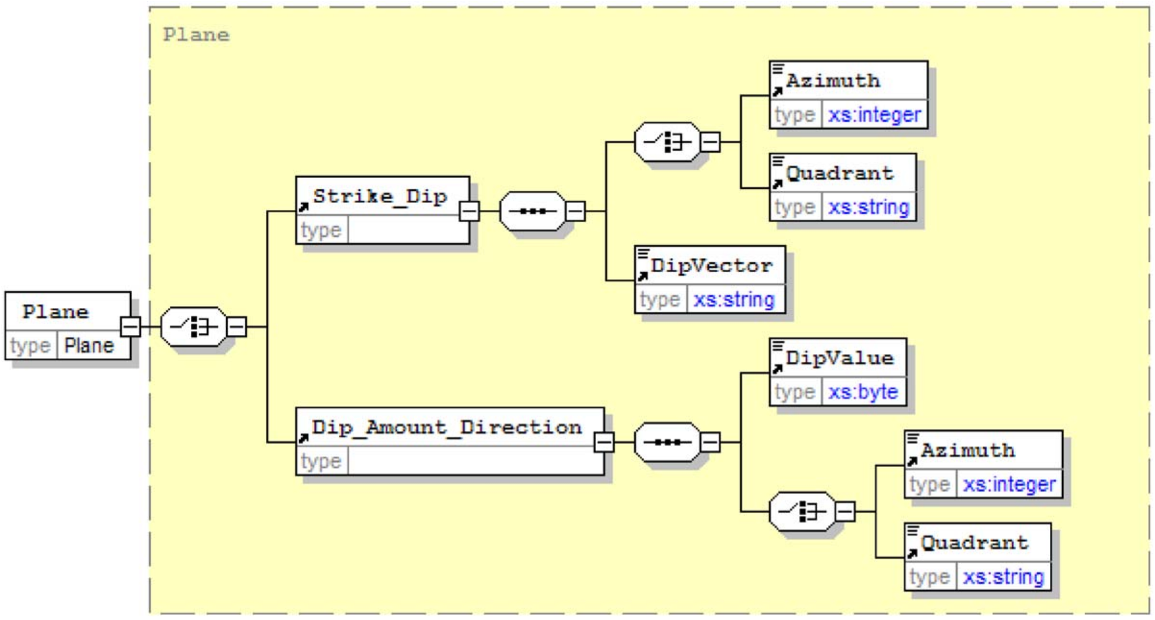


Fig. 3. XML Schema diagram showing sequence and choice compositors for Plane element reusing plane type defined in a type library. Code for Azimuth and Quadrant are given in Section 3.2.1 above. This modular code can be used for any planar geological feature such as vein, fault, bedding, and dike.

*3.2.1.1. Composition and reuse.* Complex types may have a simple or complex content. Simple content is used to define leaf node elements that only contain attributes (i.e., no child elements), and use existing simple data types through restriction or extension. The following is an example of a velocity complex type that has a simpleContent. The velocity element extends the XML Schema's xs:float type, and adds a 'direction' attribute of type string (e.g., 5.5 cm/yr):

```
<xs:complexType name="velocity">
        <xs:simpleContent>
            <xs:extension base="xs:float">
                <xs:attribute name="direction" type="xs:string"/>
            </xs:extension>
        </xs:simpleContent>
</xs:complexType>
```

Complex content is an efficient way of reusing existing user-defined complex data types through extension or restriction. The following code fragment defines a Foliation-type. This element is reusing the Plane type for the attitude of the foliation which is a planar geological feature. The XML diagram of the Plane complex type is shown in Fig. 3.

```
<xs:complexType name="Foliation_type">
    <xs:sequence>
        <xs:element name="Type" type="xs:string"/>
        <xs:element name="Attitude" type="Plane"/>
        <xs:element name="isAxialPlanar" type="xs:boolean"/>
    </xs:sequence>
  </xs:complexType>
```

The above complex type can now be reused, by extension, in the SlatyCleavage type, which "is_a" kind of Foliation. This inheritance syntax is indicated by the xs:extension base = "Foliation_type clause. The SlatyCleavage element introduces three new optional (i.e., minOccurs = "0") elements in addition to those inherited: ReductionSpot, shorteningAmount, and domain. The ReductionSpot element has an Ellipse_type type which is defined in a type library. The ShorteningAmount is a number of float type. The type library containing the Foliation_type, domain, and

Ellipse_type must be included or imported (not shown in this code fragment) in this file.

```
<xs:complexType name="SlatyCleavage">
  <xs:complexContent>
    <xs:extension base="Foliation_type">
      <xs:sequence>
        <xs:element name="ReductionSpot" type="Ellipse_type" minOccurs=""0"/>
        <xs:element name="ShorteningAmount" type="xs:float" minOccurs="0"/>
        <xs:element name="domain" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

The restriction clause is used to add constraint to an existing complex type. In the following code fragment, the Foliation complex type, defined above, is restricted by making the domain element mandatory, and adding a new mandatory microlithon element.

```
<xs:complexType name="CrenulationCleavage">
  <xs:complexContent>
    <xs:restriction base="SlatyCleavage">
      <xs:sequence>
        <xs:element name="domain" type="xs:string"/>
        <xs:element name="microlithon" type="xs:string"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
```

Reuse of elements simplifies schema composition and maintenance. Types, elements, and attributes, that are meant to be reused in several places in a schema, may be defined on the global level, i.e., on the schema level (i.e., after the document's root element), or brought into the schema through the inclusion or import syntax. Global types must be named if they are to be referred to. If global types are not meant to be instantiated, they may be declared as abstract with the abstract = "true" clause, so that other types can be derived from them through restriction and extension. For example, the code snippet above shows the reuse of the elements Ellipse_type and SlatyCleavage in the CrenulationCleavage, provided that the two types are defined somewhere at the global level, or the external schemas that define them are included.

The following schema defines the Texture global element which is reused in the Igneous element using the ref = "Texture" clause. The Igneous element has a mandatory element Name and an optional element SiO2Content, in addition to Texture which can be repeated two times. Now that Texture is globally defined, a Sedimentary or Metamorphic element (not shown in the code fragment) can reuse the same Texture element in this Rock schema. Fig. 4 depicts the schema for the Igneous element.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <!-- The Root element -->
  <xs:element name="Rock"/>
  <!-- Global elements are defined after this line -->
  <xs:element name="Texture" type="xs:string"/>
  <!-- Reuse the global element Texture (applying 'ref') in the local element Igneous -->
  <xs:element name="Igneous">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Texture" type="xs:string" maxOccurs="2"/>
        <xs:element name="Name" type="xs:string"/>
        <xs:element name="SiO2Content" type="xs:float" minOccurs="0"/>
        <!-- other elements -->
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

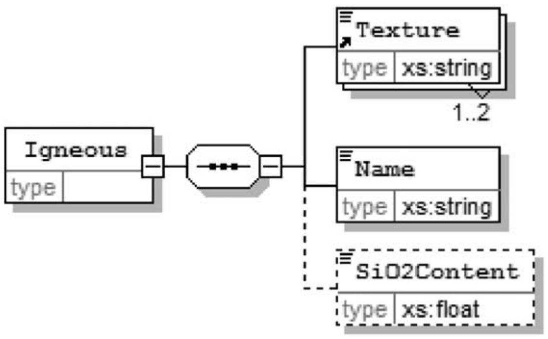


Fig. 4. XML Schema diagram for a simple Igneous element, with optional SiO2Content element.

### 3.3. Schemas and geological data

#### 3.3.1. Namespace

Integration of schemas across disciplines involves combining (reusing) heterogeneous schemas developed by different groups of people. However, this likely will lead to collision of names as the same element (tag) used in one schema may mean differently, or have a different syntax, compared with a similarly named element in another schema. The namespace mechanism in the XML Schema is created to avoid such name clashes while allowing reuse of existing schemas and avoiding redundancy. Note that namespace does not help when objects in different namespaces have different names but the same meaning.

As a general rule, we should not create a schema if a good one already exists for reuse. For example, if petrologists define a Rock schema, all other geologists are better off by reusing it instead of designing it from scratch. Hydrogeologists dealing with an aquifer, which comprises one or more formations, do not have to develop a schema for a Formation; the best is to design it in the stratigraphy namespace, and reuse it in hydrogeology.

Documents based on a schema must always use qualified elements either by using a namespace prefix or a default namespace (i.e., one without a prefix but with the same target namespace). Namespace support is provided in the XML Schema through the targetNamespace declaration as shown in the code fragment below. The target namespace may contain several schema files. One or more related schemas may have the same namespace. A document prepared based on a schema, declares the namespace of the corresponding schema, which allows the XML processor to locate the schema. The document may also have a schemaLocation clause. Using the target namespace as the default namespace is the best practice as it makes schema composition easier and much more concise. The following code fragment is a best-practice example for part of stratigraphy.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- the xmlns:xs declares a namespace prefix for the XML Schema -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   <!--declare the target namespace -->
   targetNamespace=http://www.stratigraphy.org
   <!--use target namespace as default namespace -->
   <!--the xmlns= declares no prefix, therefore it is the default namespace -->
   xmlns=http://www.stratigraphy.org
   <!--all tags now belong to the above namespace; it better be unique! -->
   elementFormDefault="qualified" attributeFormDefault="unqualified">
   <!-- include external schema and type files from the same namespace -->
   <xs:include schemaLocation="http://www.stratigraphy.org/Rock.xsd"/>
   <xs:include schemaLocation="http://www.stratigraphy.org/typeLib.xsd"/>
   <!-- import two external schemas from other namespaces -->
   <xs:import namespace="http://www.w3.org/1999/xhtml"/>
   <xs:import namespace="http://www.paleontology.org/fossils"/>
</xs:schema>
```

In the above schema, because the default namespace is the same as the declared target namespace, we do not have to prefix new elements that belong to this target namespace. The XML Schema types, however, must be prefixed with the xs (e.g., xs:string). Unqualified definitions from the included external schemas are added to the default namespace. Large schemas commonly comprise more than one file. Composing a schema involves referring to type libraries and including, importing, or redefining other external schemas. Composition of schemas must maximize the use of type libraries as they enhance reuse and help to standardize data format in the discipline. Only schemas that have the same target namespace (or have no target namespace) as the including schema can be included or redefined. The import clause can combine schemas from different target namespaces. This is very useful when combining schemas across domains (that have heterogeneous data) which naturally have different namespaces (e.g., MathML,[5] and SedML). The include, redefine, and import clauses are located at the beginning of the schema clause before element definitions.

### *3.3.2. Designing modular schemas*

XML allows breaking information into small components. Developing modular components that could be reused across domains and namespaces should be the major goal during the design of domain-specific languages. Schema reuse makes sense in geosciences because its many disciplines maintain common glossaries and concepts that can be written once and reused across the whole domain. Complex scientific concepts should be broken down into components that simplify the complexity and increase flexibility. Component-based design simplifies change as it is much easier to work with small pieces of information than the whole. Information reused across disciplines needs to be revised only once, in one schema. This saves developers and users a lot of rework and redundancy, reduces error, and simplifies the translation of documents from one language to another (Goldfarb and Prescod, 2004).

If type libraries are meant to be reused in several schemas in different namespaces, they should be designed without any target namespace as was shown in the code listing above. This way, when included in a schema that does have a target namespace, they will be added to the including schema (Daum, 2003). The following example shows this best design practice in action. The code fragment defines a small type library called Fault_typeLib.xsd which contains the type Fault_type among other types (not shown).

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
<!-- declares no target namespace -->
    <xs:complexType name="Fault_type">
     <xs:sequence>
        <xs:element name="Name" type="xs:string"/>
        <xs:element name="Length"  type="xs:float"/>
        <xs:element name="LengthUnit"  type="xs:String"/>
        <!-- other elements omitted -->
     </xs:sequence>
    </xs:complexType>
    <!-- other types -->
</xs:schema>
```

This type definition, saved in the Fault_typeLib.xsd file, can now be reused in another schema (e.g., in Earthquake.xsd) with a target namespace.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.seismology.org/earthquake"
xmlns="http://www.seismology.org/earthquake"
<!--use target namespace as default namespace -->
elementFormDefault="qualified" attributeFormDefault="unqualified">
<!--reuse another schema with the xs:include syntax -->
<xs:include schemaLocation=" Fault_typeLib.xsd "/>
    <xs:element name="Earthquake">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Name" type="xs:string"/>
                <xs:element name="Magnitude"  type="xs:float"/>
                <!-- reuse the Fault_type in a new type (Earthquake) -->
                <xs:element name="Fault"  type="Fault_type"/>
                <!-- other elements -->
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

[5]MathML, 2001. Mathematical Markup Language, http://www.w3.org/TR/REC-MathML.

Although our example type library only contains the Fault-type, for maximum modularization, a type library should contain types for all important domain objects.

## 3.4. Modular composition using type library—an example

In this section we develop the conceptual model of a small part of the plate tectonics knowledge base, and construct a reusable type library to build the main objects. A backarc basin or a mid ocean ridge (MOR) "is_a" type of divergent plate boundary. Each of the specialized sub-types of the divergent plate boundary (MOR, BackarcBasin), in addition to inheriting everything from its parent, has extra or modified characteristics, as is indicated in the class diagram for the DivergentBoundary (Fig. 5). For the sake of brevity and simplicity, only few properties of each of the objects are shown

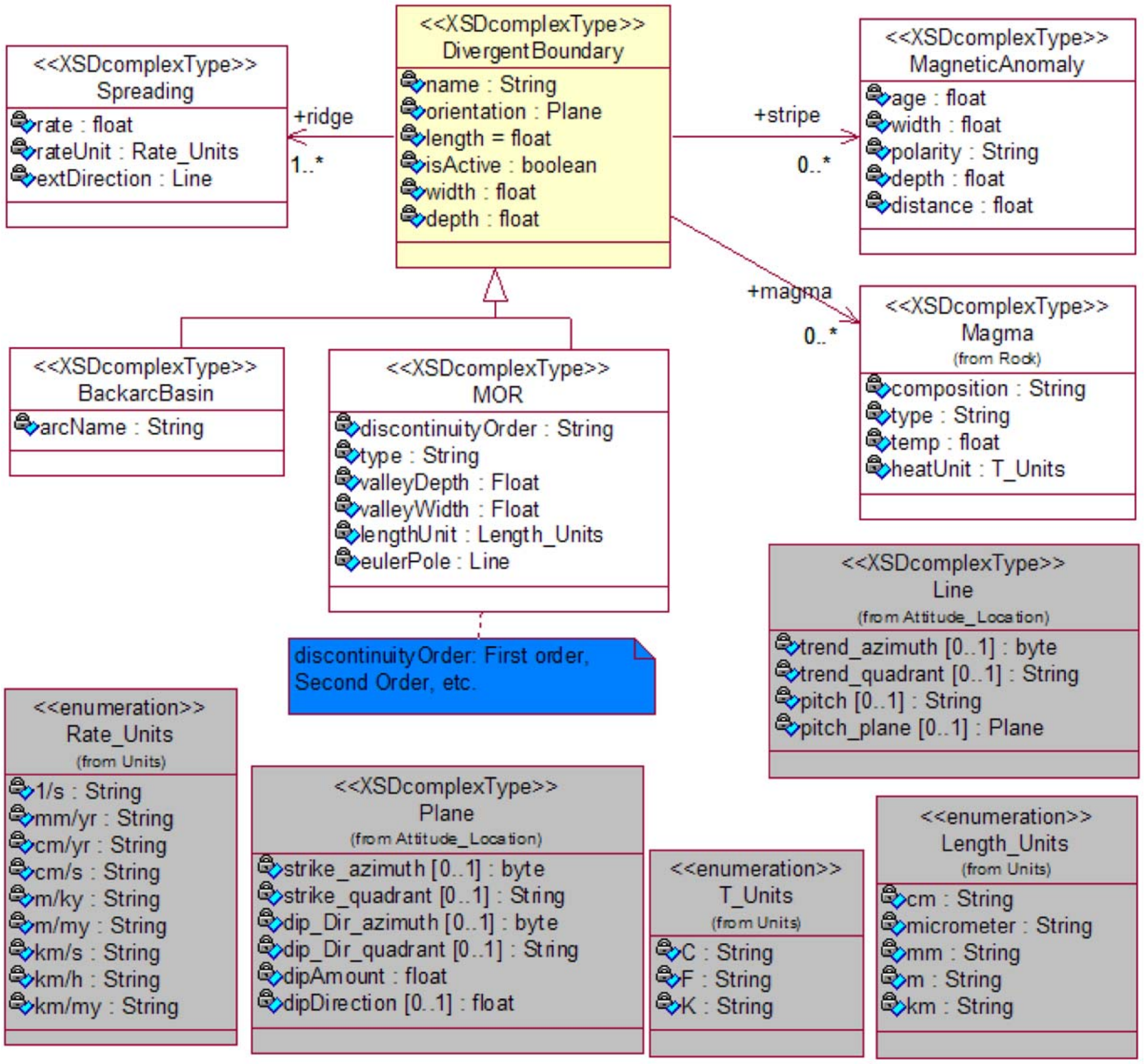


Fig. 5. UML class diagram showing a selected set of classes, with a limited set of attributes, in DivergentBoundary package of PlateTectonicsML. Shaded classes are 'included' from other packages.

in this example. The code for the simple example type library is given below, and saved in the DivergentBoundary_typeLib.xsd file.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.platetectonics.org/divergentboundary"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.platetectonics.org/divergentboundary" elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <xs:include schemaLocation="Attitude_Location_TypeLib.xsd"/>
  <!-- The Attitude_Locaiton_TypeLib.xsd file includes the Units_TypeLib.xsd -->
  <xs:annotation>
    <xs:documentation>
      Type Library for Divergent Boundary (DivergentBoundary_typeLib.xsd), defining
      reusable types.  Default namespace is the same as the target namespace,
      allowing efficient reuse.
    </xs:documentation>
  </xs:annotation>
  <!-- DivergentBoundary -->
  <xs:complexType name="DivergentBoundary_type" abstract="true">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="orientation" type="Plane"/>
      <xs:element name="length" type="xs:float"/>
      <xs:element name="lengthUnit" type="Length_Units"/>
      <xs:element name="isActive" type="xs:boolean"/>
      <xs:element name="width" type="xs:float"/>
      <xs:element name="widthUnit" type="Length_Units"/>
      <xs:element name="depth" type="xs:float"/>
      <xs:element name="depthUnit" type="Length_Units"/>
    </xs:sequence>
    <xs:attribute name="ID" type="xs:ID" use="required"/>
  </xs:complexType>
  <!-- Spreading_type -->
  <xs:complexType name="Spreading_type">
    <xs:sequence>
      <xs:element name="rate" type="xs:float"/>
      <xs:element name="rateUnit" type="Rate_Units"/>
      <xs:element name="extDirection" type="Line"/>
    </xs:sequence>
  </xs:complexType>
  <!-- MagneticAnomaly_type -->
  <xs:complexType name="MagneticAnomaly_type">
    <xs:sequence>
      <xs:element name="age" type="xs:float"/>
      <xs:element name="orientation" type="Line"/>
      <xs:element name="width" type="xs:float"/>
      <xs:element name="widthUnit" type="Length_Units"/>
      <xs:element name="polarity" type="xs:string"/>
      <xs:element name="depth" type="xs:float"/>
      <xs:element name="depthUnit" type="Length_Units"/>
      <xs:element name="distance" type="xs:float"/>
      <xs:element name="distanceUnit" type="Length_Units"/>
    </xs:sequence>
  </xs:complexType>
  <!-- Magma_type -->
  <xs:complexType name="Magma_type">
    <xs:sequence>
      <xs:element name="composition" type="xs:string"/>
      <xs:element name="type" type="xs:string"/>
      <xs:element name="temp" type="xs:float"/>
      <xs:element name="heatUnit" type="T_Units"/>
    </xs:sequence>
  </xs:complexType>
  <!-- MOR_type -->
  <xs:complexType name="MOR_type">
    <xs:complexContent>
```

```
          <xs:extension base="DivergentBoundary_type">
            <xs:sequence>
              <xs:element name="discontinuityOrder" type="xs:string"/>
              <xs:element name="type" type="xs:string"/>
              <xs:element name="valleyDepth" type="xs:float"/>
              <xs:element name="depthUnit" type="Length_Units"/>
              <xs:element name="valleyWidth" type="xs:float"/>
              <xs:element name="widthUnit" type="Length_Units"/>
              <xs:element name="eulerPole" type="Line"/>
               </xs:sequence>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
        <!-- BackarcBasin_type -->
        <xs:complexType name="BackarcBasin_type">
          <xs:complexContent>
            <xs:extension base="DivergentBoundary_type">
              <xs:sequence>
                <xs:element name="arcName" type="xs:string"/>
              </xs:sequence>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:schema>
```

The XML diagram in Fig. 6 shows how the DivergentBoundary.xsd schema reuses the type library (DivergentBoundary_typeLib.xsd). The code for the DivergentBoundary.xsd file is given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.platetectonics.org/divergentboundary"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.platetectonics.org/divergentboundary" elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <!-- Divergent Boundary -->
  <xs:include schemaLocation="DivergentBoundary_TypeLib.xsd"/>
  <xs:element name="DivergentBoundary">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="DivergentBoundary_type">
          <xs:sequence>
            <xs:element ref="Spreading" maxOccurs="unbounded"/>
            <xs:element ref="MagneticAnomaly" maxOccurs="unbounded"/>
            <xs:element ref="Magma" minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <!-- Spreading -->
  <xs:element name="Spreading">
   <xs:complexType>
     <xs:complexContent>
       <xs:extension base="Spreading_type"/>
     </xs:complexContent>
   </xs:complexType>
 </xs:element>
```

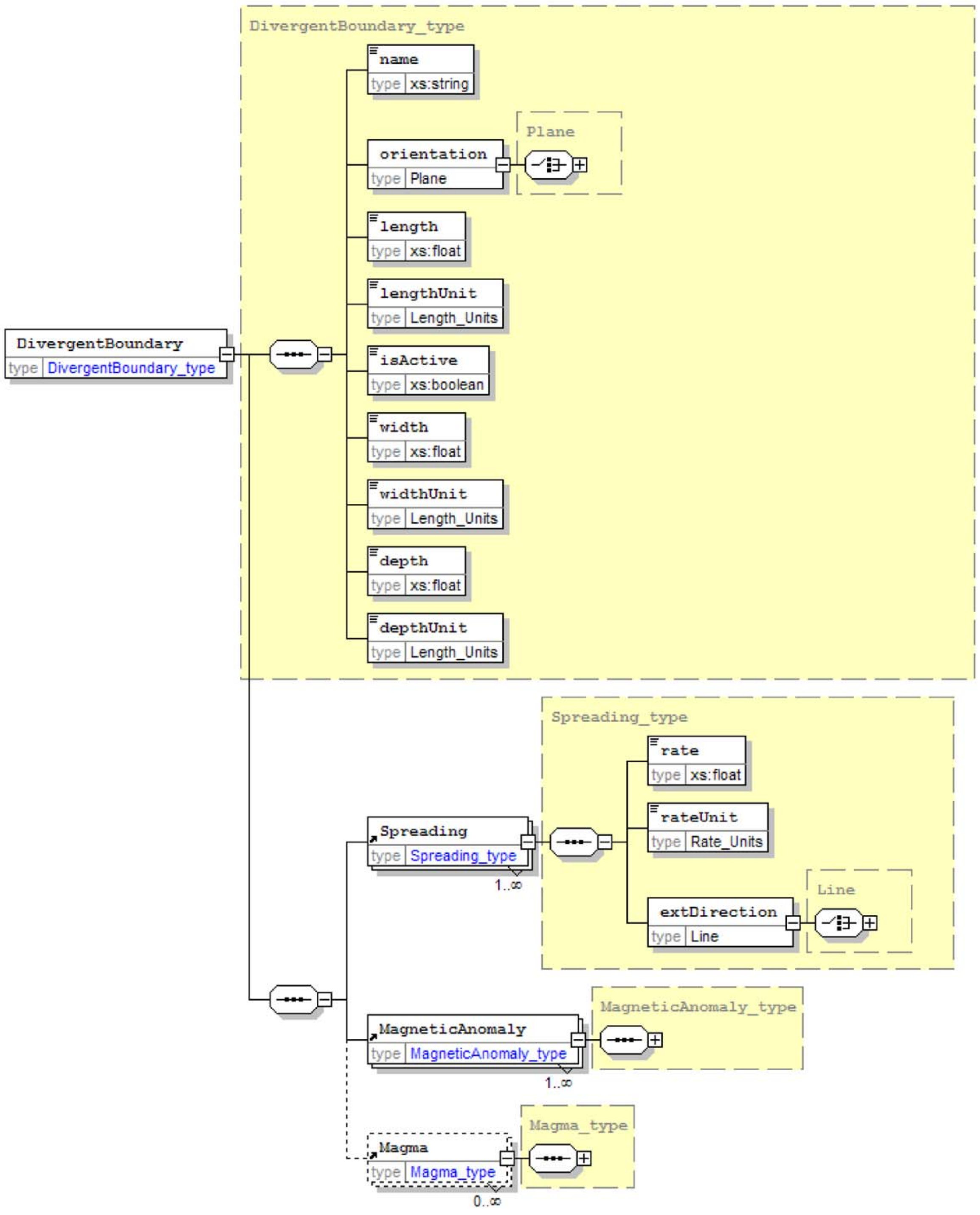


Fig. 6. XML Schema diagram showing composition of DivergentBoundary elements by reusing DivergentBoundary_typeLib.xsd type library. See Fig. 3 for the Plane type.

```
    <!--MagneticAnomaly -->
    <xs:element name="MagneticAnomaly">
        <xs:complexType>
            <xs:complexContent>
                <xs:extension base="MagneticAnomaly_type"/>
            </xs:complexContent>
        </xs:complexType>
    </xs:element>
    <!--Magma -->
    <xs:element name="Magma">
        <xs:complexType>
            <xs:complexContent>
                <xs:extension base="Magma_type"/>
            </xs:complexContent>
        </xs:complexType>
    </xs:element>
    <!-- MOR -->
    <xs:element name="MOR">
        <xs:complexType>
            <xs:complexContent>
                <xs:extension base="MOR_type"/>
            </xs:complexContent>
        </xs:complexType>
    </xs:element>
    <!-- BackarcBasin_type-->
    <xs:element name="BackarcBasin">
        <xs:complexType>
            <xs:complexContent>
                <xs:extension base="BackarcBasin_type"/>
            </xs:complexContent>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

## 4. Summary

Efficient data interchange over the Web requires capturing and translating the traditional vocabulary and non-standard, data structures into precise and structured, domain-specific markup languages. These standard languages allow us to give precise definition and meaning to geological terms, and assign syntax, data structure, and type to each scientific concept. The languages minimize the loss of information in transit from one source to another, and allow efficient sharing, storage, and management of information.

The XML Schema's namespace and other reuse and composition mechanisms allow building domain-specific schemas from existing schemas within and across field, applying the inclusion and import syntaxes, respectively. The XML Schema can readily be mapped into relational database schema (Babaie and Babaei, 2005), allowing a one-to-one correspondence between the data storage and the markup language. We provided XSD code for a subset of plate tectonics using the principles discussed in this paper.

## Acknowledgments

Hassan Babaie wishes to thank Georgia State University's Research Initiation Grant for funding this project. The SEED Program of the Rational Software Company and now IBM's Scholar Program generously provided free copies of the Rational Rose Enterprise Edition to Hassan Babaie which was used in conceptual modeling. The authors thank Mike Wright and Ron Lake for their critical review which led to improvement of the manuscript.

## References

Babaie, H., Babaei, A., 2003. Development of the plate tectonics and seismology markup languages with XML. EGS-AGU-EUG Joint Assembly, abstract; 11 April 2003, Nice, France.

Babaie, H., Babaei, A., 2005. Developing the Earthquake Markup Language and database with UML and XML schema. Computers & Geosciences, this issue, doi:10.1016/j.cageo.2004.12.010.

Binstock, C., Peterson, D., Smith, M., Wooding, M., Dix, C., Galtenberg, C., 2003. The XML Schema Complete Reference. Addison-Wesley, Boston, MA 965pp.

Booch, G., Rumbaugh, J., Jacobson, I., 1999. The Unified Modeling Language User Guide. Addison-Wesley, Boston, MA 482pp.

Carlson, D., 2001. Modeling XML Applications with UML: Practical e-Business Applications. Addison-Wesley, Boston, MA 333pp.

Daum, B., 2003. Modeling Business Objects with XML Schema. Morgan Kaufman Publishers, Heidelberg, Germany 535pp.

Eeles, P., Houston, K., Kozaczynski, W., 2002. Building J2EE Applications with the Rational Unified Process. Addison-Wesley, Boston, MA 265pp.

Gillette, J.E., 2002. A practical framework for understanding KM. In: Bellaver, R.F., Lusa, J.M. (Eds.), Knowledge Management Strategy and Technology. Artech House, Boston, MA, pp. 1–22.

Goldfarb, C.F., Prescod, P., 2004. XML Handbook, fifth ed. Prentice Hall, PTR, Upper Saddle River, NJ, pp. 1222.

Kähre, J., 2002. The Mathematical Theory of Information. Kluwer Academic Publishers, Boston, MA 502pp.

Larman, C., 2002. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, second ed. Prentice Hall PTR, Upper Saddle River, NJ 627pp.

Rumizen, M.C., 2002. The Complete Idiot's Guide to Knowledge Management. Alpha; A Pearson Publication Company, 315pp.

Van der Vlist, E., 2002. XML Schema. O'Reilly & Associates, Sebastopol, CA 379 pp.

Watson, I., 2003. Applying Knowledge Management: Techniques for Building Corporate Memories. Morgan Kaufman Publishers, Amsterdam 252pp.