Information and Software Technology 45 (2003) 445–460

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Modeling imprecise requirements with XML

Jonathan Lee[*], Yong-Yi Fanjiang

Department of Computer Science and Information Engineering, National Central University, Chungli 32054, Taiwan

Abstract

Fuzzy theory is suitable to capture and analyze the informal requirements that are imprecise in nature, meanwhile, XML is emerging as one of the dominant data formats for data processing on the internet. In this paper, we have developed a fuzzy object-oriented modeling technique (FOOM) schema based on XML to model requirements specifications and incorporated the notion of stereotype to facilitate the modeling of imprecise requirements. FOOM schema is also transformed into a set of application programming interfaces (APIs) in an automatic manner. A schema graph is proposed to serve as an intermediate representation for the structure of FOOM schema to bridge the FOOM schema and APIs for both content validation and data access for the XML documents.

Keywords: XML; Fuzzy object; Imprecise requirements

1. Introduction

One of the foci of the recent developments in object-oriented modeling (OOM) has been the extension of OOM to fuzzy logic to capture informal requirements that are imprecise in nature (see Ref. [10] for a survey on fuzzy object-oriented model). Meanwhile, XML is emerging as one of the dominant data formats for data processing on the Internet [20]. XML is rapidly establishing itself as the metagrammar for interorganizational communication and becoming increasingly urgent that requirements analysts, system designer and software developers be able to: (1) model the information represented in XML, and (2) describe the relationships between the XML and the systems to process it.

In this paper, we propose (an overview of our approach is depicted in Fig. 1):

- To define a fuzzy object-oriented modeling technique (FOOM) [11] schema for modeling the FOOM requirements specifications in XML format: as a continuation of our previous work in using fuzzy logic as a basis for formulating imprecise requirements [12], we have extended FOOM along two dimensions:

1. To define the FOOM schema for constructing requirements specifications, and for validating the model: the FOOM schema is defined based on the key features described in FOOM, including fuzzy set, fuzzy attribute, fuzzy rule, and fuzzy association, by using the XML schema; and
2. To incorporate the notion of stereotypes in FOOM to facilitate the modeling of imprecise requirement: we extend the FOOM by incorporating three kinds of stereotype: entity, control, and interface. In addition, we also add a new stereotype, fuzzy entity to better describe the semantics of imprecise requirements.

- To transform the FOOM schema into a set of application programming interfaces (APIs) for content validation and data access in an automatic manner: a schema graph is used to serve as an intermediate representation for the structure of FOOM schema to bridge the FOOM schema and APIs for both content validation and data access for an XML document. Through the APIs generated from the schema graph, the XML documents are parsed to perform the structure and content validation, and to access the data from an object tree. The object tree is a tree-like structure for an XML document as a result of the parsing. A top-level document instance is the root of the tree with zero or more element objects as its children. An element object is defined in a recursive manner with zero or more attributes and/or text elements. In an element object, there are two kinds of methods: get and set method (see Fig. 2). An object tree serves as an internal representation of an XML document. APIs can exploit

[*] Corresponding author. Tel.: +886-3-422-715 ext. 4510; fax: +886-3-422-2681.

E-mail addresses: yjlee@selab.csie.ncu.edu.tw (J. Lee), yyfanj@selab.csie.ncu.edu.tw (Y.Y. Fanjiang).

0950-5849/03/$ - see front matter © 2003 Elsevier Science B.V. All rights reserved.
doi:10.1016/S0950-5849(03)00015-6

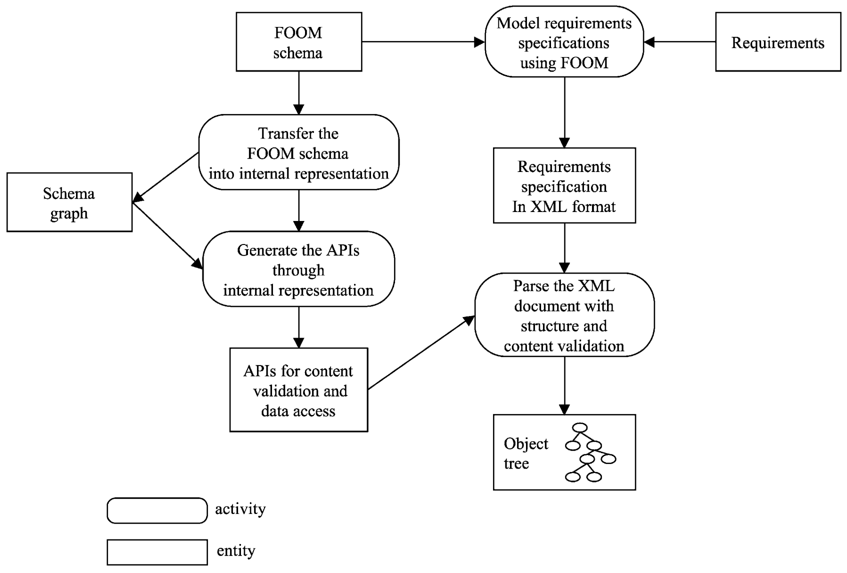

FOOM schema
Model requirements specifications using FOOM
Requirements
Transfer the FOOM schema into internal representation
Schema graph
Requirements specification In XML format
Generate the APIs through internal representation
Parse the XML document with structure and content validation
APIs for content validation and data access
Object tree
activity
entity

Fig. 1. An overview of the proposed approach.

the information in the object tree to perform the content validation and access the data of an XML document.

We chose the meeting schedule problem [23] as an example throughout this paper to illustrate the proposed approach.

The organization of this paper is as follow. We first introduce the background on XML and XML schema in Section 2. In Section 3, the mappings of FOOM to XML schema and to XML documents are discussed. Several kinds of fuzziness in fuzzy objects are identified, and an extension of FOOM with stereotypes is also described. In Section 4, we propose a schema graph as an intermediate representation for an XML schema, and algorithms for transforming the XML schema into a set of APIs especially for performing the structure and content validation and data access of XML documents. The implementation of FOOM prototype is briefly described in Section 5. Related works are discussed in Section 6, and concluding remarks are given in Section 7.

2. Extensible markup language (XML)

XML [4] is a data description language standardized by the World Wide Web Consortium (W3C). XML is a sophisticated subset of SGML, and designed to describe document data using arbitrary tags. One of the goals of XML is to be suitable for use on the Web. As its name implies, extensibility is a key feature of XML; users or applications are free to declare and use their own tags and attributes. Therefore, XML ensures that both the logical structure and content of semantics rich information are retained. XML emphasizes description of information structure and content as distinct from its presentation. The logical structure of an XML document comprises of properly nested XML elements. An XML element has a name, may have attributes, and a content model. An element is encoded syntactically with a start tag, which includes the element's attributes, the content and the end tag. The tag is made up of the element's name. Only three data types are supported for attributes, i.e. character data, special purpose XML tokenized types, and enumerated types. XML has a global namespace and does not support inheritance. The declarative part of XML, which defines the XML elements, their attributes and how they are structured, is called a Document Type Declaration or DTD, which is derivative from SGML and defines a series of tags and their constraints.

The W3C XML schema [2,6,21] is a language for defining the structure of XML document instance that belongs to a specific document type. XML schema can be seen as replacing the XML DTD syntax. XML schema

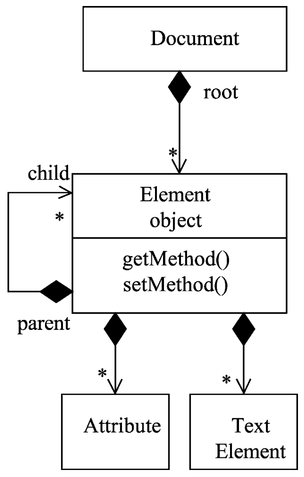


Fig. 2. The structure of an object tree.

provides strong data typing, modularization, and reuse mechanisms not available in XML DTD. It provides the necessary framework for creating XML documents by specifying the valid structure, constraints, and data types for various elements and attributes of an XML document. The XML schema specification defines several different built-in data types, such as string, integer, boolean, date, and time, among others. The specification also provides the capability for defining new types through the use of complex type and simple type. There is a basic difference between complex types, which allow elements in their content and may carry attributes, and simple types, which cannot have element content and cannot carry attributes.

New complex types are defined using the ⟨complexType⟩ tag and such definition typically contains a set of element declarations, element references, and attribute declarations. The declarations are not themselves types, but rather an association between a name and the constraints, which govern the appearance of that name in documents restricted by the associated schema. Elements are declared using the ⟨element⟩ tag, and attributes are declared using the ⟨attribute⟩ tag. In general, an element is required to appear when the value of ⟨minOccurs⟩ is one or more. The maximum number of times an element may appear is determined by the value of a ⟨maxOccurs⟩ attribute in its declaration. This value may be a positive integer, or the term *unbounded* to indicate there is no maximum number of occurrences. The default value for both the ⟨minOccurs⟩ and the ⟨maxOccurs⟩ attributes is one. Thus, when an element is declared without a ⟨maxOccurs⟩ attribute, the element may not occur more than once. Similarly, if user specifies a value for only the ⟨maxOccurs⟩ attribute, it must be greater than or equal to the default value of ⟨minOccurs⟩. If both attributes are omitted, the element must appear exactly once.

New simple types are defined by deriving them from existing simple types (built-in's and derived). In particular, a user can derive a new simple type by restricting an existing simple type, in other words, the legal range of values for the new type are a subset of the existing type's range of values. The ⟨simpleType⟩ tag is used to define and name the new simple type, as well as the ⟨restriction⟩ tag is used to indicate the existing (base) type and to identify the facets that constrain the range of values.

Developers can use those built-in as well as user-defined data types to effectively define and constrain attributes and element values in an XML document. The XML schema requirements are divided into three groups: structural, data type, and XML conformance requirements. The structure requirements include support for inheritance and constraints on structural constructs. The data type requirements calls for primitive data types, well-defined lexical representation, and support for user-defined data types. These are further explained below:

- Structure. XML schema structure specifies the XML schema definition language, which offers facilities for describing the structure and constraining the contents of XML documents, including those which exploit the XML namespace facility.
- Data type. It defines facilities for defining data types to be used in XML schema as well as other XML specifications. The data type language, which is itself represented in XML, provides a superset of the capabilities found in XML DTD for specifying data types on elements and attributes.
- Reuse. Another key feature of XML schema language is that it supports inheritance. We can create new schema by deriving features from when new ones are required. The XML schema language also provides for breaking a schema into separate components. We can then refer to appropriate predefined components in writing schema. Inheritance enables efficient software reuse and helps developers avoid building everything from scratch. It significantly improves XML software development process, code maintainability, and programmer productivity.

3. Mapping fuzzy object oriented model to XML schema

FOOM [11] is a modeling approach to analyzing imprecise requirements which extends the traditional OOM along several dimensions: (1) to extend a class to a fuzzy class which classifies objects with similar properties, (2) to encapsulate fuzzy rules in a class to describe the relationship between attributes, (3) to evaluate fuzzy class memberships by considering both static and dynamic properties, and (4) to model uncertain fuzzy associations between classes.

3.1. Fuzzy classes

Several kinds of fuzziness that are required to model imprecise requirements are identified:

- classes with imprecise boundary to describe a group of objects with similar attributes, similar operations and similar relationships (Fig. 3);
- rules with linguistic terms that are encapsulated in a class to describe the relationships between attributes;

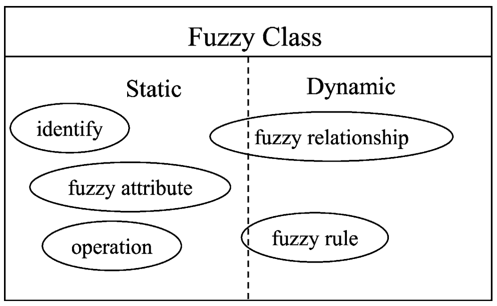


Fig. 3. Properties encapsulated in a fuzzy class.

- ranges of an attribute with linguistic values or typical values in a class to define the set of allowed values that instances of that class may take for the attribute;
- the membership degree (i.e. ISA degree) between an object and a class, and between a subclass and its super class (i.e. AKO degree) can be mapped to the interval [0,1]; and
- associations between classes that an object instance may participate to some extent.

FOOM uses fuzzy inclusion technique to compute the compatibility degree between a class and an object, and the class membership between a class and its subclass (i.e. perceptual fuzziness). Instead of using fuzzy inclusion technique to clustering data, the focus is to determine the compatibility between objects based on their properties (i.e. attribute, operation, and association).

3.2. Enhancing FOOM with stereotypes

We have further extended FOOM by incorporating three kinds of stereotypes [8]: entity, control, and interface. An entity class models information in a system that should be held for a long time, and should typically survive a use case. All behavior naturally coupling to this entity should also be placed in the entity class. An interface class models behavior and information that is dependent on the interface to a system. A control class models functionality that is not naturally tied to any other classes. Typically such behavior consists of operating on several different entity classes, doing some computations and then returning the result to an instance object.

A new stereotype, *fuzzy entity*, is also defined to better capture imprecise requirements. A fuzzy entity class is a kind of entity class with similar attributes, operations and relationships. Let us take the meeting schedule system as an example (see Fig. 4), there are two interface classes, three control classes, and four entity classes. The interface classes describe the input and output forms. The control classes specify the control behavior that is not naturally tied to any other classes, for example, the create participant class specifies the behavior of adding participants and registering meeting. Since the degree that a person belongs to the class *Important Participant* depends on his *status* and his *role* in the meeting he attends, the Important Participant is specified as a fuzzy entity class.

3.3. FOOM schema

In order to model imprecise requirements with XML, we have formulated the FOOM schema based on the key features defined in FOOM: fuzzy set, fuzzy attribute, fuzzy rule, and fuzzy association, by means of the XML schema. The FOOM schema serves as a vehicle to better describe the syntax and semantics of the imprecise requirements.

The basic element of the fuzzy term is the fuzzy set described by a membership function. There are two kinds of fuzzy set: discrete and continuous. A discrete fuzzy set is a collection of objects. Each object has an attached degree of membership. In the case of a normalized membership function, the degree of membership ranges from 0.0 to 1.0. For a continuous fuzzy set, we adopt the definition in Ref. [22] to assume the curve that a continuous fuzzy set's membership function can be approximated to with connected points. To each point, we assign a unique degree of membership as a real number, called f-value. The degree of membership between the specified f-values is calculated by interpolation.

Both types of fuzzy sets are captured using the XML schema (see Fig. 5 and 6). Fig. 5 specifies the XML schema definition of the discrete fuzzy sets. It consists of zero or more pairs of the elements ⟨object⟩ and ⟨membership-degree⟩. An element of type ⟨object⟩ can be character data or any child elements, which is specified as ⟨xsd:anyType⟩ in XML schema. The content of the element ⟨membership-degree⟩ stands for the range of this value between 0.0 and 1.0. We specify the ⟨membership-degree⟩ as a float data type defined in XML schema, and restrict this data type's range with ⟨maxInclusive⟩ in 1.0, and ⟨minInclusive⟩ in 0.0. The continuous fuzzy set is specified using XML schema in Fig. 6. Continuous fuzzy set as well as fuzzy numbers and intervals can be described by a set of points, which are defined as a ⟨complexType⟩ element with a pair of the elements ⟨f-value⟩ and ⟨membership-degree⟩. Fig. 7 shows an example of a continuous fuzzy set.

A fuzzy attribute is mapped to a pair of the element ⟨name⟩ and ⟨fuzzy-range⟩ (see Fig. 8). The ⟨name⟩ tag can be any string to represent the attribute's name. A fuzzy attribute in FOOM is the ranges with linguistic values or typical values in a class to define the set of allowed values that an instance of that class may take. We define the attribute's fuzzy range to be a set of linguistic-values or typical-values (see Fig. 8). Each linguistic-value is characterized by a fuzzy set. Each typical-value includes a tag of ⟨t-degree⟩, which is an kind of membership-degree type. Fig. 9 is an example of a fuzzy attribute in XML format with the typical-value named status, consisting of the values student, staff and faculty.

Using the fuzzy rules is one way to deal with imprecision where a rule's conditional part and/or the conclusion part contains linguistic variables. A fuzzy rule is mapped to a sequence of ⟨if⟩ elements and ⟨then⟩ elements. The ⟨if⟩ element contains more than one ⟨condition⟩ element declaration with a content model that is a sequence of the ⟨variable⟩, ⟨operator⟩ and ⟨statements⟩ element. The ⟨statement⟩ element type also contains either a ⟨value⟩ element or a sequence of ⟨value⟩, ⟨connector⟩ and ⟨statements⟩ element. A fuzzy rule mapping to an XML schema is shown in Fig. 10. Fig. 11 gives an example of using XML to markup a fuzzy rule.

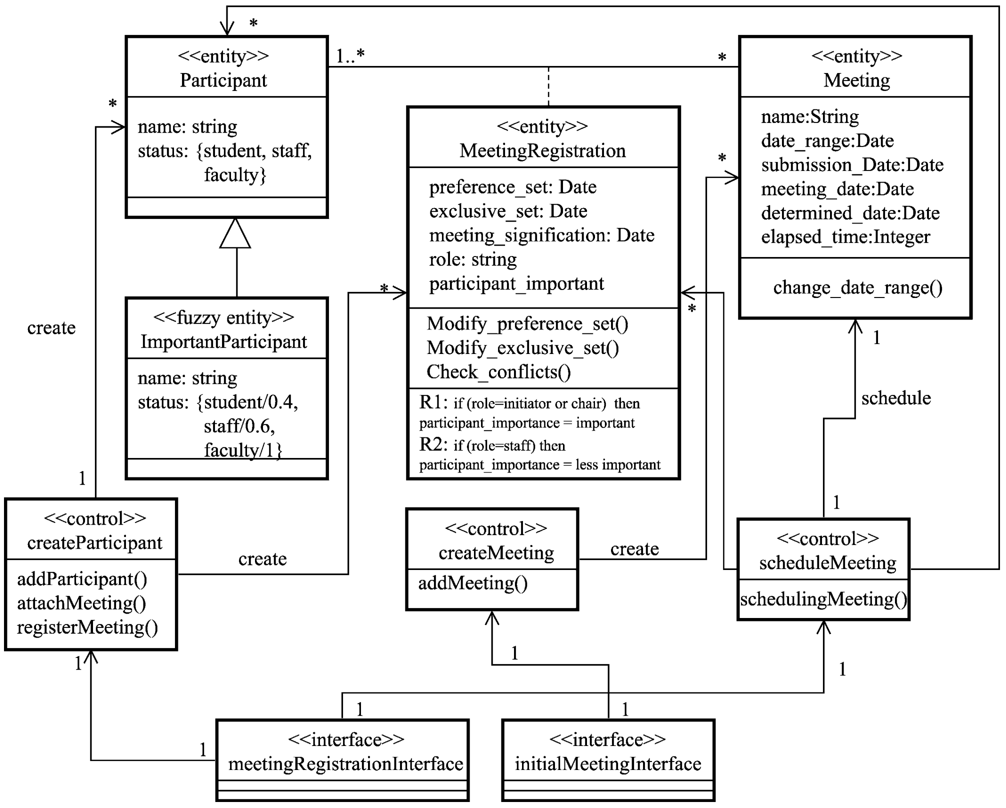


Fig. 4. An example of meeting schedule system using FOOM.

A fuzzy association is mapped to an association type which is a complex type. The content model of an association is a sequence of ⟨description⟩, ⟨link-attribute⟩, ⟨association-end⟩, ⟨association-end⟩, ⟨degree-of-participation⟩ and ⟨possibility-degree⟩ elements. The value that a link takes for the link attribute is described by the degree of participation to represent the degree that a link participates in this association. The ⟨possibility-degree⟩ is a confidence level of this fuzzy association, whose value is a fuzzy truth value. Fig. 12 shows an example of an XML schema for fuzzy association.

4. Transforming an XML schema to APIs

In this section, we will discuss the transformation from FOOM schema into a set of content validation and data access APIs through a schema graph. The schema graph is an extension of DTD graph [17] with typing information to serve as an intermediate representation for describing the structure of an XML schema.

4.1. Schema graph

A schema graph is a directed, typed, bipartite graph consisting of four kinds of nodes: element, text element, attribute, and operator, and arcs are directed links from a node to another one representing the containable relationship.

- An element node has a name and type information. Each element node has zero or more children, which can be an element, text element, attribute, or operator node.
- The text element and attribute nodes have a name, typing information, and constraints on the type. The typing information can be string characters or other data types defined in an XML schema (e.g. float). The element and text element nodes have an associated set of attribute nodes.

```
<xsd:complexType name="discrete-fuzzy-setType">
  <xsd:sequence>
    <xsd:group ref="objects" minOccurs= "0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:group name="objects">
  <xsd:sequence>
    <xsd:element name="object" type="xsd:anyType"/>
    <xsd:element name="membership-degree" type="membership-degreeType"/>
  </xsd:sequence>
</xsd:group>

<xsd:simpleType name="membership-degreeType>
  <xsd:restriction base="xsd:float"/>
   <xsd:maxInclusive value="1"/>
   <xsd:minInclusive value="0"/>
  </xsd:restriction>
</xsd:simpleType>
```

Fig. 5. XML schema of a discrete fuzzy set.

```
<xsd:complexType name="fuzzy-setType">
  <xsd:sequence>
    <xsd:element name="point" type="pointType" minOccurs= “0” maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="pointType”>
  <xsd:sequence>
    <xsd:element name=“f-value” type=“xsd:anyType”/>
    <xsd:element name=“membership-degree” type=“membership-degreeType”/>
  </xsd:sequence>
</xsd:complexType>
```

Fig. 6. XML schema of a continuous fuzzy set.

- The operator node is to represent the following operators: ‘*’ (set with zero or more elements), ‘ + ’ (set with one or more elements), ‘?’ (optional), and ‘|’ (or) specified in an XML schema.

A formal definition 1 of schema graph is delineated below.

Definition 1. A schema graph (SG) is defined as a six-tuple,

$$SG = (E, TE, AN, O, A, R) \tag{1}$$

$E = \{e_i(t_i)|i = 1...n\}$ is a finite set of element nodes, where t_i represents the type of the element node E_i. $TE = \{te_j(t_j, c_j)|j = 1...m\}$ is a finite set of text element nodes, where t_j is to represent the type of the text element node te_j, and c_j represents the constraint of type t_j. $AN = \{an_k(t_k, c_k)|k = 1...p\}$ is a finite set of attribute nodes, where t_k represents the type of the attribute node an_k, and c_k represents the constraint of type t_k. $O = \{o_l(t_l)|l = 1...q\}$ is a finite set of operator nodes, where t_l represents the type of the operator node o_l. $A \subseteq (E \times (E \cup TE \cup AN \cup O)) \cup (O \times (E \cup TE \cup AN \cup O))$ is a set of arcs. $R \subseteq \{E \cup TE\}$ is a set of the root nodes in this schema graph.Fig. 13 is an example of graphic depiction of the schema graph representing the fuzzy attribute schema in Fig. 8. In Fig. 13, there are six element nodes, four text element nodes, three attribute nodes, and four operator nodes. Each node, except the operator node, has a name and corresponding type information. For example, the linguistic value node is an element node, and its type is the same as its name. The membership degree node is a text element, and its type is float. In addition, the text element nodes and attribute nodes have both type and constraint. For example, the membership degree node’s type is float with value ranging between 0 and 1.

Definition 2. Given a schema graph SG, let the $e \in E$, we define $SN = AllSubNode(e)$, where $SN = \{sn_i|i = 1...n\} \subseteq \{E \cup TE \cup AN \cup O\}$, if the SN is the set of all directly sub nodes of e in SG.

Definition 3. Given a schema graph SG, let the $e \in E$, we define $LN = LeftmostSubNode(e)$, where $LN \in \{E \cup TE \cup O\}$, if the LN is the leftmost directly sub element, text element or operator node of e in SG.

Definition 4. Given a schema graph SG, let the $e \in E$, we define $LNO = LeftmostNonOperatorNode(e)$, where

```
<fuzzy-set>
  <point>
    <f-value> 150 </f-value>
    <membership-degree> 0.0 </membership-degree>
  </point>
  <point>
    <f-value> 165 </f-value>
    <membership-degree> 1.0 </membership-degree>
  </point>
  <point>
    <f-value> 175 </f-value>
    <membership-degree> 1.0 </membership-degree>
  </point>
  <point>
    <f-value> 190 </f-value>
    <membership-degree> 0.0 </membership-degree>
  </point>
</fuzzy-set>
```

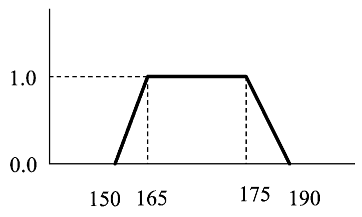
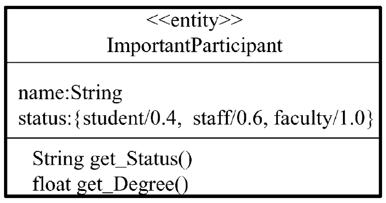


Fig. 7. An example of continuous fuzzy set.

```
<xsd:complexType name=“fuzzy-attributeType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name=“fuzzy-range" type=“fuzzy-rangeType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name=“fuzzy-rangeType">
  <xsd:choice>
    <xsd:group ref="linguistic-valueGroup"/>
    <xsd:group ref="typical-valueGroup"/>
  </xsd:choice>
</xsd:complexType>

<xsd:group name = “linguistic-valueGroup”>
  <xsd:sequence>
   <xsd:element name=“linguistic-value” type=“linguistic-valueType” maxOccus=“unbounded”/>
  </xsd:sequence>
</xsd:group>

<xsd:group name = “typical-valueGroup”>
  <xsd:sequence>
    <xsd:element name=“typical-value” type=“typical-valueType” maxOccus=“unbounded”/>
  </xsd:sequence>
</xsd:group>

<xsd:complexType name="linguistic-valueType">
  <xsd:sequence>
    <xsd:element name="fuzzy-set" type="fuzzy-setType"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="typical-valueType">
  <xsd:sequence>
    <xsd:element name="t-degree" type="membership-degreeType"/>
  </xsd:sequence>
  <xsd:attribute name=“name" type="xsd:string" use="required"/>
</xsd:complexType>
```

Fig. 8. XML schema of fuzzy attributes.



```
<fuzzy-attribute>
  <name> status </name>

  <fuzzy-range>

    <typical-value name=“student”>
      <t-degree> 0.4 </t-degree>
    </typical-value>

    <typical-value name=“staff”>
      <t-degree> 0.6 </t-degree>
    </typical-value>

    <typical-value name=“faculty”>
      <t-degree> 1.0 </t-degree>
    </typical-value>

  </fuzzy-range>

</fuzzy-attribute>
```

Fig. 9. An example of fuzzy class in XML format with the fuzzy attribute.

```
<xsd:complexType name="fuzzy-ruleType">
  <xsd:sequence>
    <xds:element name="if" type="ifType"/>
     <xsd:element name="then" type="thenType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ifType">
  <xsd:sequence>
    <xsd:element "condition" type="conditionType"/>
   </xsd:sequence>
</xsd:complexType>

 <xsd:complexType name="conditionType">
   <xsd:sequence>
      <xsd:element name="variable" type="variableType"/>
       <xsd:element name="operator" type="operatorType"/>
       <xsd:element name="statement" type="statementType"/>
    </xsd:sequence>
    <xsd:attribute name="property"
          type="propertyType" use="required"/>
 </xsd:complexType>

 <xsd:simpleType name="variableType">
   <xsd:extension base="xsd:anyType"/>
 </xsd:simpleType>

<xsd:simpleType name="propertyType"/>
  <xsd:restriction base ="xsd:string">
    <enumeration value="fuzzy-attribute"/>
    <enumeration value="attribute"/>
  </xsd:restriction>
</xsd:simpleType>
```

```
<xsd:complexType name="statementType">
  <xsd:choice>
    <xsd:element name="value" type="valueType"/>
    <xsd:group ref ="statementgroup">
  </xsd:choice>
</xsd:complexType>

<xsd:group name="statementgroup">
  <xsd:sequence>
    <xsd:element name="value" type="valueType"/>
     <xsd:element name="connector" type="connectorType"/>
     <xsd:element name="statement" type="statementType"/>
  </xsd:sequence>
</xsd:group>

<xsd:complexType name="thenType">
  <xsd:sequence>
    <xsd:element name="variable" type=variableType"/>
    <xsd:element name="assignment" type="assignmentType"/>
    <xsd:element name="hudges" type="hudgeType"/>
    <xsd:element name="fuzzy-set" type="fuzzy-setType"/>
  </xsd:sequence>
  <xsd:attribute name="property"
         type="propertyType" use="required"/>
</xsd:complexType>
```

Fig. 10. The XML schema for fuzzy rules.

$LNO \in \{E \cup TE\}$, if the LNO is the leftmost directly sub element or text element node skipping operator node of e in SG.

Definition 5. Given a schema graph SG, let the $e \in \{E \cup TE \cup O\}$, $N = \{n_j \in AllSubNode(e) | j = 1...m\}$, we define $ConditionNodes(e)$ as below:

1. *if* $e \in \{E \cup TE\}$,

   $ConditionNodes(e) = \{e\}$

2. if $e \in O$ and e's type is '*' or ' + ' or '?', let $n_r = LeftmostNonOperatorNode(e) \in N$, where $1 \le r \le m$

   $$ConditionNodes(e) = \bigcup_{k=1...r} ConditionNodes(n_k), \quad n_k \in N$$

3. if $e \in O$ and e's type is '|',

   $$ConditionNodes(e) = \bigcup_{j=1...m} ConditionNodes(n_j), \quad n_j \in N$$

Definition 6. Let $n \in O$ in G, we define $SubNodeOfOperation(n) = \{x_i | x_i \in E \cup TE \cup AN\}$ as below:

for all $s \in AllSubNode(n)$,
if $s \in \{E \cup TE \cup AN\}$, then $s \in SubNodeOfOperation(n)$
else if $s \in O$, then $SubNodeOfOperation(n) = SubNodeOfOperation(n) \cup SubNodeOfOperation(s)$

If (role = initiator or chair or secretary)
then participant important = more important

```
<fuzzy-rule>
  <if>
    <condition>
       <variable> role </variable>
       <operator> = </operator>
     <statement>
        <value> initiator </value>
        <connector> or </connector>
        <statement>
          <value> chair </value>
          <connector> or </connector>
          <value> secretary </value>
        </statement>
       </statement>
    </condition>
  </if>
  <then>
     <variable property= "fuzzy-attribute"> participant important </variable>
     <assignment/>
      <hudges>
          <modifier> more </modifier>
      <hudges>
      <fuzzy-set name="important" />
  </then>
</fuzzy-rule>
```

Fig. 11. An example of a fuzzy rule.

```
<xsd:complexType name="AssociationType">
  <xsd:sequence>
    <xsd:group ref="description"/>
   <xsd:element name="link-attribute" type="nameType" minOccurs="0"/>
   <xsd:element name="association-end" type="association-endType"/>
   <xsd:element name="association-end" type="association-endType"/>
   <xsd:element name="degree-of-participant" type="membership-degreeType" minOccurs="0"/>
   <xsd:element name="possibility-degree" type="possibility-degreeType" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

Fig. 12. The XML schema for a fuzzy association.

4.2. Transforming an XML schema into a schema graph

The schema graph is used to serve as an intermediate representation for the structure of an XML schema to bridge the XML schema and APIs for both content validation and data access for an XML document. The main idea is to construct the four types of nodes: element, text element, attribute, and operator, by parsing an XML schema. Element nodes are built for the elements declared by the ⟨complexType⟩ tag. The attribute and text element nodes are established based on the content model of the elements. Operator nodes are constructed by examining the occurrences of elements in the content model. We formally describe the details on how to establish a schema graph from an XML schema in Algorithm 1.

Algorithm 1. (Generating a schema graph from an XML schema)

```
1. create a null graph $G$ and set point ← null;
2. for each simpleType $S_j$ element declared in XML schema
   (a) construct the mapping table between the simpleType's name and it's type and constraint information for the constructing of the text element or attribute nodes.
3. for each complexType element $E_i$ declared in XML schema
   (a) create an element node $e_i$ in $G$;
   (b) set point ← $e_i$;
   (c) if this complexType consists a ⟨choice⟩ tag
       (i) create a operator node $o_i$('|');
       (ii) create an arc from point to $o_i$;
       (iii) point ← $o_i$
   (d) for all the attribute $AN_l$ declared in $E_i$
       (i) create an attribute node $an_l$;
       (ii) find this node's type and constraints infor-
```

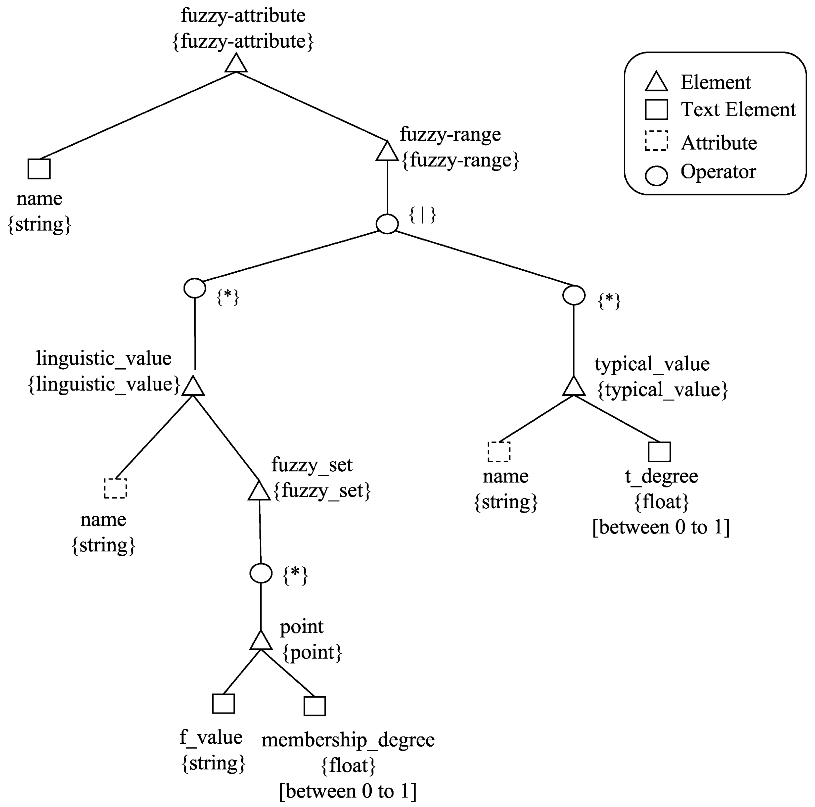


Fig. 13. A schema graph for a fuzzy attribute schema.

```
            mation from the mapping table;
        (iii) assign the type and constraints information to
            $an_l$ node;
        (iv) create an arc from point to $an_l$
    (e) for all element $E_k$ declared in the content mode of
        $E_i$
    (f) if $E_k$'s type is complexType then create a element
        node $e_k$; else if $E_k$'s type is simpleType then
        create a text element node $te_k$, and assign it the
        type and constraint information form the mapping
        table;
        (i) if $E_k$'s number of instance are more than
            zero, then
            (•) create a operator node $o_k$('*');
            (•) create two arcs from point to $o_k$ and
        from $o_k$ to $e_k$ or $te_k$;
        (ii) else if $E_k$'s number of instance are more than
            one, then
            (•) create a operator node $o_k$('+');
            (•) create two arcs from point to $o_k$ and
        from $o_k$ to $e_k$ or $te_k$;
        (iii) if $E_k$'s number of instance is one or zero, then
            (•) create a operator node $o_k$('?');
            (•) create two arcs from point to $o_k$ and
        from $o_k$ to $e_k$ or $te_k$;
    (g) for each element $E_k$ declared in $E_i$
        (i) set point $\leftarrow e_k$;
        (ii) repeat the step 3 until no element has been
            declared in the content model of $E_i$
```

4.3. Generating APIs for content validations and data access

The content validation APIs are used to validate whether a constructed XML document is consistent with the corresponding XML schema definition. The data access APIs are used to get and set the data contained in an XML document. Those APIs can parse an XML document into an object tree for content validation and data access in the XML documents.

We have defined three basic rules below to construct the content validation APIs.

- for the '*' node, apply the *while loop* statement,
- for the ' + ' node, apply the *do while* statement, and
- for the '|' node, apply the *if then else* statement.

The key idea is to locate the element nodes in a schema graph and to construct classes with attributes, constructor, and methods which can then be used for content validation and data access. Algorithms 2 and 3 show the details on how the APIs are generated.

Algorithm 2. (Generating APIs for XML validation and data access)

```
input: schema graph $SG$, output: a set of APIs for XML
document validation and data access.
for each root node $r_i \in R$ in $SG$ processes the following
steps, until all the root nodes are visited.

1. set the current node ($v$) as $r_i$.
2. if $v$ is not visited
    (a) if $v \in \{TE \cup AN\}$ then set $r$ is visited and goto
        step 1.
    (b) if $v \in O$ then set v as visited and for each node in
        $AllSubNode(v)$ to repeat the step 2.
    (c) if $v \in E$ then
        (i) to create a class named by $v$'s name
        (ii) to generate a set of attributes, constructor and
            operations of this class by using Algorithm 3
            with taking $v$ as input.
        (iii) to set v as visited, and for each node in
            $AllSubNode(v)$ to repeat the step 2.
```

Algorithm 3. (Creating the contents of Class)

```
input: $r \in \{E \cup TE \cup AN \cup O\}$, output: a set of attributes,
constructor and operations statements in a class.
for all $s_j \in AllSubNode(r)$

1. if $s_j \in E$
    (a) to insert an attribute in this class named _sj and
        declare it's type as $s_j$ class;
    (b) to apply the if-statement to check if it's data type
        is correct and validation according to the schema
        graph;
    (c) to insert an operation [public $s_j$ get$S_j$(); ] and an
        operation [public void setget$S_j$($s_j$ s); ]
2. if $s_j \in \{TE \cup AN\}$
    (a) to insert an attribute in this class named _sj and
        declare it's type as $s_j$'s type;
    (b) to apply the if-statement to check if it's data type
        is correct and validation according to the schema
        graph;
    (c) to insert an operation [public $s_j^j$s type get$S_j$ (); ]
        and an operation [public void set $S_j$ ($s_j^j$s type s); ]
3. if $s_j \in O$
    (a) if $s_j$'s type is '*'
        i. to apply the while-loop-statement to check if
            it is validation. The condition part of the
            while-loop-statement is the set of
            ConditionNodes($s_j$).
        ii. for all $ss_k \in SubNodeOfOperator(s_j)$
            A. to insert an attribute in this class named
                _$ss_k$ and declare it's type as Vector;
            B. to insert an operation [public Vector
                get $SS_j$ (); ] and an operation [public
                void set$SS_j$ (Vector ss); ]
    (b) if $s_j$'s type is ' + '
        i. to apply the do-while-statement to check if it
            is validation. The condition part of the
            do-while-statement is the set of
```

ConditionNodes(s_j).

ii. for all $ss_k \in SubNodeOfOperator(s_j)$

A. to insert an attribute in this class named *_ssk* and declare it's type as Vector;

B. to insert an operation [*public Vector getSS*$_j$ ();] and an operation [*public void setSS*$_j$ (*Vector ss*);]

(c) if s_j's type is '|'

i. to apply the *if-then-elseif-statement* to check if it is validation. The condition part of the *if-then-elseif-statement* is the set of *ConditionNodes*(s_j).

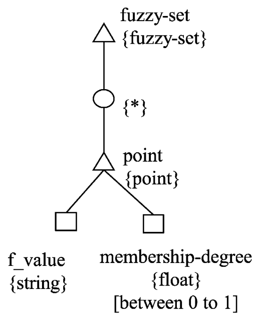
ii. to recall this algorithm with taking s_j as it's input.

(d) if s_j's type is '?'

(i) to apply the if-statement to check if it is validation. The condition part of the *if-statement* is the set of *ConditionNodes*(s_j).

(ii) recall this algorithm with taking s_j as it's input.

In the following, a part of the schema graph in Fig. 13—the fuzzy sets schema, is used to illustrate the transformation from the schema graph into its corresponding APIs (see Fig. 14). We first generate a fuzzy_set class containing an attribute: _point. Notice that the fuzzy_set contains a set of point elements, therefore, the type of this attribute is declared as a vector. We then construct a while loop statement to check to see if the XML document has a ⟨point⟩ element tag. If it is true, a new point object will be constructed and added into the _point vector as a new element. Finally, the data access APIs (get and set methods) can be established. Similar to the fuzzy_set class, the ⟨point⟩ element consists of a pair of the element ⟨f-value⟩ and ⟨membership-degree⟩, therefore, we will build a Point class with two attributes: _f_value and membership_degree that are typed string and float, respectively. In the constructor statement of the Point class, we first check to see if the XML document has a ⟨f-value⟩ element tag. If it is true, we assign the value of this element tag to _f_value, else return an invalid message. In addition to checking the existence of



```
class fuzzy_set {
  //attribute
  Vector _point;

  //constructor
  fuzzy_set() {
    while (element.getTag()=="point") {
        _point.add(new Point());
        element=element.getNextElement();
    }
  }

  //operation
  public Vector getPoint() {return _point;}
  public void setPoint(Vector point) {this._point = point;}
}
class Point {
  //attribute
  String _f_value;
  float _membership_degree;

  //constructor
  Point() {
    if (element.getTag()=="f-value") {
      this._f_value = element.getValue();
      element=element.getNextElement();
    } else {Error("This is not a validated XML docuement");}
    if (element.getTag()=="membership-degree") {
      float f = Float.parseFloat(element.getValue());
      if (f >=0 && f <=1)
        {this._f_degree = f;}
      else {Error("This is not a validated XML docuement");}
      element=element.getNextElement();
    } else {Error("This is not a validated XML docuement");}
  }

  //operation
  public String getF_value() {return this._f_value;}
  public String getMembership_degree()
    {return this._membership_degree;}
  public void setF_value(String f_value)
    {this._f_value = f_value;}
  public void setMembership_degree(float membership_degree)
    { this._membership_degree = membership_degree; }
}
```

Fig. 14. The schema graph of fuzzy sets and it's corresponding APIs.

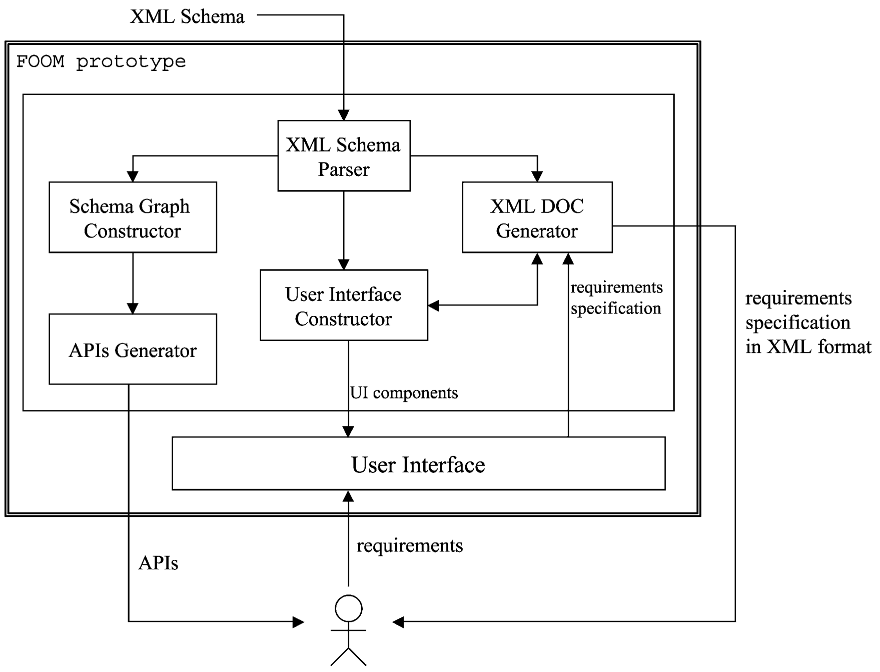


Fig. 15. An overview of FOOM prototype.

the element tag ⟨membership-degree⟩, we also need to check to see if the value of this element is consistent with the constraints specified in the schema graph before a new value can be assigned to the _f_value attribute. Finally, we construct the data access APIs for the Point class by get and set methods.

5. Implementation

We adopted Java as the programming language for the FOOM prototype (see Fig. 15). In this prototype, a user can use the UML-like notation [3,15] to model and document user's requirements with a specific XML schema (i.e. FOOM schema). Basic notations of FOOM (e.g. class, relationship, fuzzy-AKO, fuzzy-ISA, etc.) are provided for describing the specification. The user can construct the object/class diagram and specify the internal characteristics (e.g. fuzzy constraints, fuzzy rules, linguistic value or typical value, etc.) of the target system. There are three main parts in this prototype. The first one is the XML Schema Parser and User Interface Constructor that parse the input schema to construct the corresponding user interface (UI) components. The UI components are UML-like notations with fuzziness features, which can be used to model the imprecise requirements. The second part is the XML Document Generator that constructs the XML documents according to the requirements specifications and FOOM schema. The final part is the Schema Graph Constructor and APIs Generator that translate the XML schema into schema graph and corresponding APIs, respectively. Fig. 16 is an illustration of the object/class diagram of the meeting schedule system. Corresponding APIs can also be generated in this prototype (see Fig. 17).

6. Related work

XML is a data format for structured document interchange on the Web. It provides a framework for tagging structured data by allowing developers to define an unlimited set of tags to bring great flexibility. In general, XML serves three different sorts of role in the extant approaches:

6.1. Serving as a data exchange format

J. Suzuki and Y. Yamamoto [19] proposed an exchange format for UML models based on XML, called UXF (UML eXchange Format). It is a format powerful enough to express, publish, access and exchange UML models and a natural extension from the existing Internet environment. It serves as a communication vehicle for developers, and as a well-structured data format for development tools. UXF is an application of XML and designed to be flexible enough to encode and exchange any UML constructs. UXF facilitates intercommunications between software developers, interconnectivity between development tools, and natural extension from existing Web environments.

The XML Metadata Interchange Format (XMI) [14] provides a standard way to exchange information about metadata between modeling tools based on the united modeling language (UML) object-based modeling language. XMI specifies an open information interchange model that is intended to give developers working with

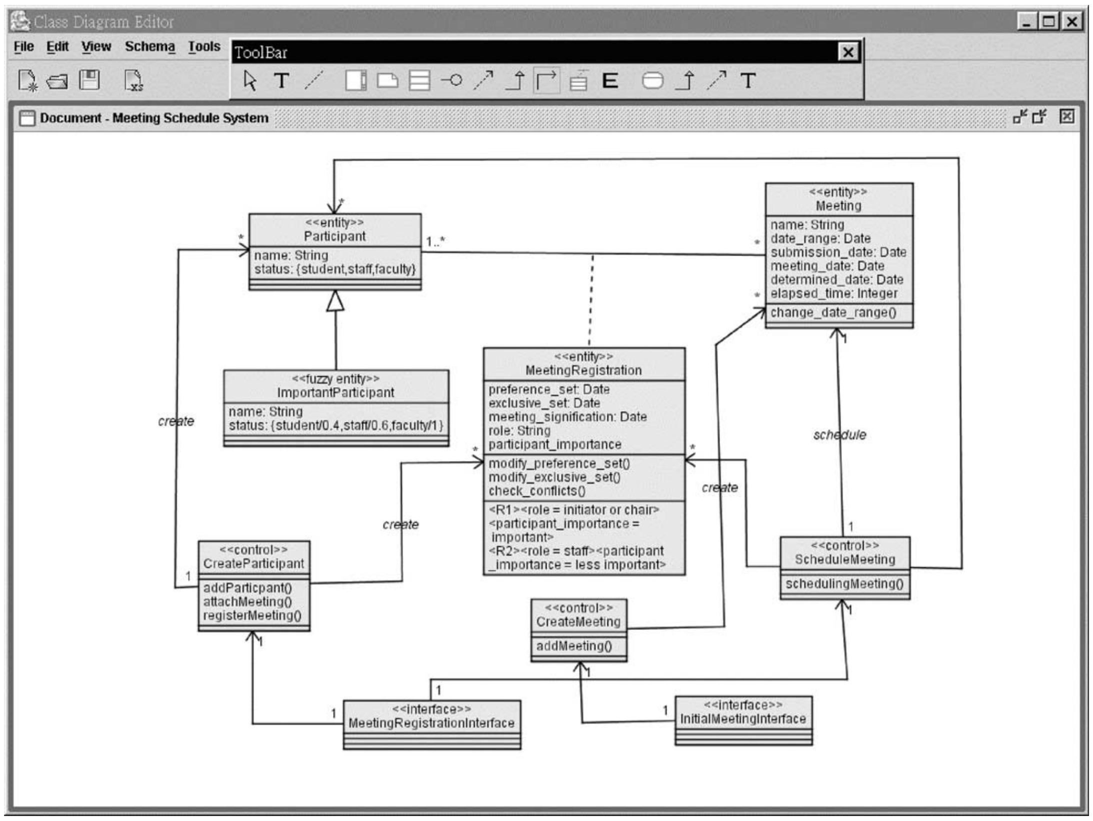

Class Diagram Editor
File Edit View Schema Tools
ToolBar
Document - Meeting Schedule System
<<entity>> Participant
name: String
status: {student,staff,faculty}
<<fuzzy entity>> ImportantParticipant
name: String
status: {student/0.4,staff/0.6,faculty/1}
<<entity>> Meeting
name: String
date_range: Date
submission_date: Date
meeting_date: Date
determined_date: Date
elapsed_time: Integer
change_date_range()
<<entity>> MeetingRegistration
preference_set: Date
exclusive_set: Date
meeting_signification: Date
role: String
participant_importance
modify_preference_set()
modify_exclusive_set()
check_conflicts()
<R1><role = initiator or chair> <participant_importance = important>
<R2><role = staff><participant _importance = less important>
<<control>> CreateParticipant
addParticpant()
attachMeeting()
registerMeeting()
<<control>> CreateMeeting
addMeeting()
<<control>> ScheduleMeeting
schedulingMeeting()
<<interface>> MeetingRegistrationInterface
<<interface>> InitialMeetingInterface
create
create
create
schedule
1..*
*
1

Fig. 16. An example in FOOM prototype.

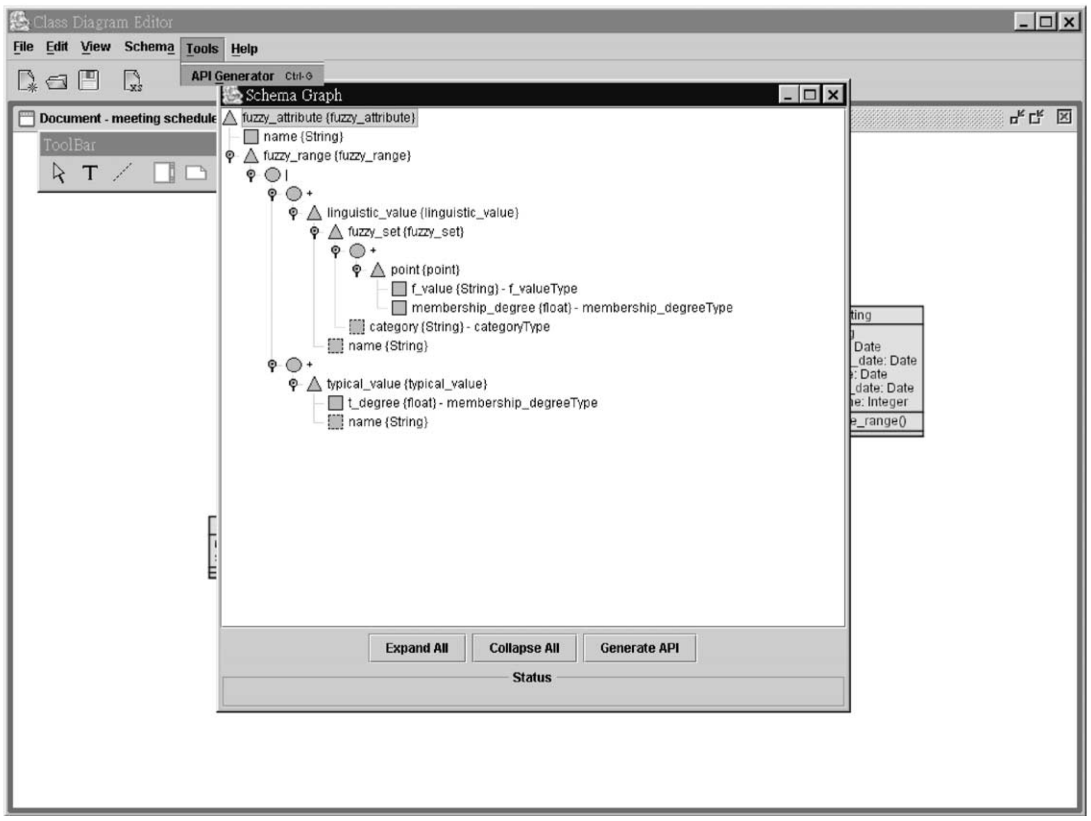

Class Diagram Editor
File Edit View Schema Tools Help
API Generator Ctrl-G
Schema Graph
Document - meeting schedule
ToolBar
fuzzy_attribute {fuzzy_attribute}
name {String}
fuzzy_range {fuzzy_range}
|
+
linguistic_value {linguistic_value}
fuzzy_set {fuzzy_set}
+
point {point}
f_value {String} - f_valueType
membership_degree {float} - membership_degreeType
category {String} - categoryType
name {String}
+
typical_value {typical_value}
t_degree {float} - membership_degreeType
name {String}
Expand All
Collapse All
Generate API
Status

Fig. 17. An example of schema graph generation.

object technology the ability to exchange programming data over the Internet in a standardized way, thus bringing consistency and compatibility to applications created in collaborative environments. XMI is a much-needed specification to bring consistency and compatibility to applications created in collaborative environments. XMI is an open, stream-based interchange format formed by the integration of three key industry standards:

- XML: A W3C standard that defines an open, metamodel-neutral, programming language-neutral, API-neutral, streamable, textual, human-readable format to bring structured information to the Web.
- UML:Unified Modeling Language, an OMG modeling standard that defines a rich, object-oriented modeling language/notations for object-oriented analysis and design (OA&D).
- MOF: Meta Object Facility, an OMG meta modeling and metadata repository standard that specifies an extensive framework for defining models for metadata and represents it as CORBA objects; uses UML notations for models.

D. Skogan has proposed an approach using UML as a schema language for XML based data interchange [18]. The mapping idea is to use UML class and package diagrams and map the various concepts to XML elements. The mapping described is inspired by the Object Management Group's (OMG) XML Metadata Interchange (XMI), but is simplified in the sense that only the static and organizational parts of UML's metamodel are mapped (package, class, attribute, and association). The CORBA data types have been defined that may reduce the number of characters in the element names.

K. Turowski and U. Weng [22] introduce a formal syntax for important fuzzy data types used to store fuzzy information. They make this syntax operable by defining appropriate document type definition (DTD), and show how fuzzy information, whose description is based on these DTDs, can be exchanged between application systems by means of the XML.

6.2. Serving as a mediation for software artifacts transformation

A. Muller et al. [13] provide an approach using XML as the basic technology to semi-automatically drive user interfaces based on an abstract definition of the UI and a description of input–output device capabilities. In order to automate matching an abstract user interface with the capabilities of a concrete target platform and device, they define a device-independent representation of the user interaction components called XML based User Interface Description (UID) which is defined by DTD. A mapping function between abstract user interface objects stored in the UID and concrete UI elements contained in the XML-based device definition is also defined.

D.H. Park and S.D. Kim [16] propose the code generator and mapping rule descriptor to define the relationship between UML class and various kinds of programming source code. The proposed rule provides higher level constructs to the developer for describing the way of code generation. By making the code generator independent of repository format, the increase of the applicability of the code generator is shown. Mapping rule descriptor is a XML document that describes how to translate model data to source code. This document consists of one header and several generations. Header part contains self-descriptive contents links as document name, used language, author and so forth. Each generation part contains relationship between model data and source code, and an organization and several structural elements to describe rules for generating a file.

6.3. Serving as an intermediate representation for database systems

Relational database systems have been used in a variety of systems for storing XML data, generally by either storing the data as a large string object, or by mapping the XML schema onto the relational schema template. A number of researches have reported progress towards the storing of the XML documents in the relational database [1,5,7,17,24], which can be classified into two categories:

Designing database schema with DTDs: J. Shanmugasundaram et al. [17] propose an approach to analyzing DTD and automatically converting it into relational schemas. In their approach, a DTD is simplified by discarding the information on the order of occurrence among elements. Thus, the simplified DTD preserves only the semantics of child elements concerned with whether the element (a) can occur only once or more times, and (b) is mandatory or not. A graph, called DTD graph, based on the simplified information is proposed to serve as the intermediate representation for transforming the DTD into corresponding database schema. In this approach, nodes with an in-degree of one are inlined in the parent node's relation. For each element node with an in-degree of zero, a separate relation is created because they are not reachable from any other node. In the DTD graph, edges marked with '*' indicate that the element of a destination node can occur more than once. For each such element, a separate relation is created because relational databases cannot store set values as they are. Finally, element nodes, which appear along with the directed paths from the element in the DTD graph that creates the relational schema, are also inlined as an attribute in the relational technique.

Designing database schema without DTDs: M. Yoshikawa et al. [24] propose an approach to the storage and retrieval of XML documents that uses relational database, called XRel. They employ the XPath data model to represent XML documents. In the XPath data model, XML documents are modeled as an ordered tree. Based on XPath, they cut down the nodes of Xpath into four types: root node, element node, attribute node, and text node. For each node except the root node, they store the information on the path from the root node to other nodes. For processing the XML query, they present an algorithm for translating a core subset of Xpath expressions into SQL query. C. Berkley et al. [1] design a schema-independent data storage system for the XML called Metacat. In Metacat project, an XML document is structured as a tree of nodes where the root node is the document entity and children of the root node are elements and attributes. Metacat uses the DOM object model to store XML documents in a relational database, parses the XML document into a series of DOM nodes, and then inserts each node as a record into a database table. The table has a defined recursive foreign key, which allows each record to point to it's parent. D. Florescu and D. Kossmann [7] assume an XML document can be represented as an ordered and labeled directed graph. Each XML element is represented as a node in the graph; the node is labeled with the *oid* of the XML object. Element–subelement relationships are represented by edges in the graph and labeled by the name of the subelement. In their proposed approach, there are three ways to store the edges of a graph into relational database. The simplest scheme is to store all edges of the graph that represents an XML document in a single table, called *Edge* table. The second mapping scheme is to group all edges with the same label into one table. This method resembles the binary storage scheme proposed to store semi-structured data. They create as many Binary tables as different subelement and attribute names occur in the XML document. The third approach is to generate a single *Universal* table to store all the edges. This *Universal* table corresponds to the result of a full outer join of all Binary tables.

Our approach differs from the previous researches in the following aspects:

- Managing imprecise requirements by means of XML schema; and
- Deriving a set of corresponding APIs based on the XML schema in an automatic manner for content validation and data access.

7. Conclusion

In this paper, we have proposed: (1) defining the FOOM schema for modeling the FOOM requirements specifications in XML format, as well as incorporating the notion of stereotypes to facilitate the modeling of imprecise requirements; and (2) transforming the FOOM schema into a set of APIs through the use of the schema graph as an intermediate representation for content validation and data access in an automatic manner.

In our approach, the FOOM schema provides a useful representation for modeling the imprecise requirements. Through the FOOM schema defined by means of the XML schema, the developer can model their precise requirements, transfer the requirements into an XML document, and then use the rendering mechanism (e.g. XSL [9]) to transform the XML document into software artifacts, including the source codes. Moreover, the APIs that are automatically derived based on the XML schema can make easy the development of an XML parser.

Our future research plan will focus on the defuzzification of the XML-based imprecise requirements specifications into their corresponding crisp code templates.

Acknowledgements

We thank Mr. Ying-Yan Lin for his help in implementing the prototype. This research is supported by National Science Council (Taiwan) under grants NSC90-2213-E-008-031 and NSC91-2213-E-008-012.

References

[1] C. Berkley, M. Jones, J. Bojilova, D. Higgins, Metacat: a schema-independent XML database system, Proceedings of the International Conference on Scientific and Statistical Database Management July (2001) 171–179.
[2] P.V. Biron, A. Malhotra, XML Schema part 2: datatypes, W3C Recommendation, May 2001, http://www.w3.org/TR/xmlschema-2/.
[3] G. Booch, J. Eumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison Wesley, Reading, MA, 1999.
[4] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, Extensible Markup Language (XML) 1.0 (second edition), W3C Recommendation, October 2000, http://www.w3.org/TR/REC-xml/.
[5] A. Deutsch, M. Fernandez, D. Suciu, Storing semistructured data with STORED, Proceedings of the ACM SIGMOD Conference on Management of Data May (1999).
[6] D.C. Fallside, XML Schema part 0: primer, W3C Recommendation, May 2001, http://www.w3.org/TR/xmlschema-0/.
[7] D. Florescu, D. Kossmann, Storing and querying XML data using an RDMBS, IEEE Data Engineering Technology Bulletin 22 (3) (1999).
[8] I. Jacobson, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, Reading, MA, 1992.
[9] M. Kay, XSL Transformations (XSLT) version 2.0, W3C Working Draft, December 2001, http://www.w3.org/TR/xslt20/.
[10] J. Lee, J.Y. Kuo, N.L. Xue, A note on current approaches to extending fuzzy logic to object-oriented modeling, International Journal of Intelligent Systems 16 (7) (2001) 807–820.
[11] J. Lee, N.L. Xue, K.H. Hsu, S.J. Yang, Modeling imprecise requirements with fuzzy objects, Information Sciences 118 (1999) 101–119.
[12] J. Lee, N.L. Xue, J.Y. Kuo, Structuring requirement specifications with goals, Information and Software Technology 43 (2001) 121–135.
[13] A. Mueller, T. Mundt, W. Lindner, Using XML to semi-automatically derive user interfaces, Proceedings of the Second

International Workshop on User Interfaces to Data Intensive Systems (2001).

[14] OMG, XML Metadata Interchange (XMI)—proposal to the OMG OA&DTF RFP 3: Stream-based Model Interchange Format (SMIF), Technical Report AD Document AD/98-10-05, Object Management Group, m492 Old Connecticut Path, Framingham, MA 01701, USA, 1998.

[15] OMG, Unified modeling language specification, version 1.4, June 1999, http://www.omg.org/.

[16] D.H. Park, S.D. Kim, XML rule based source code generator for UML CASE tool, The Eight Asis-Pacific Software Engineering Conference (APSEC 2001) December (2001) 53–60.

[17] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, J. Naughton, Relational databases for querying xml documents: limitations and opportunities, Proceedings of the 25th International Conference on Very Large Data Bases (1999).

[18] D. Skogan, UML as a schema language for XML based data interchange, Proceedings of the second International Conference on The Unified Modeling Language (UML'99) (1999).

[19] J. Suzuki, Y. Yamamoto, Managing the software design documents with XML, ACM SIGDOC Conference 16 (1998).

[20] J. Suzuki, Y. Yamamoto, Toward the interoperable software design models: quartet of UML, XML, DOM and CORBA, Proceedings of the Fourth IEEE International Symposium and Forum on Software Engineering Standards (1999).

[21] H.S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, XML Schema part 1: structures, W3C Recommendation, May 2001, http://www.w3.org/TR/xmlschema-1/.

[22] K. Turowski, U. Weng, Representing and processing fuzzy information—an XML-based approach, Knowledge-Based Systems 15 (1–2) (2002) 67–75.

[23] A. van Lamsweerde, R. Darimont, P. Massonet, Goal-directed elaboration of requirements for a meeting scheduler problems and lessons learnt, Technical Report RR-94-10, Universite Catholique de Louvain, Departement d'Informatique, B-1348 Louvain-la-Neuve, Belgium, 1994.

[24] M. Yoshikawa, T. Amagasa, T. Shimura, S. Uemure, XRel: a path-based approach to storage and retrieval of XML documents using relational databases, ACM Transactions on Internet Technology 1 (1) (2001).