A Formal Specification of G-DTD: A Conceptual Model to Describe XML Documents

Zurinahni Zainol*, Bing Wang*  
Department of Computer Science  
University of Hull  
UK  
School of Computer Sciences  
Universiti Sains Malaysia  
Penang, Malaysia  
z.zainol@2007.hull.ac.uk, b.wang@hull.ac.uk

Abstract – This paper provides a formal specification in Z of a conceptual model for an XML document called Graph-Document Type Definition (G-DTD). This model has been used for describing XML documents at the schema level and also assists the user to arrange the content of XML documents. More importantly G-DTD can be used as a tool to simplify the XML document design in a simple and precise way. The specification presented here provides a formal account of the state and operation of this model and a sound basis for instantiations of the model to be built.

Keywords – XML model and design; graphical notation; DTD; formal methods

I. INTRODUCTION

It is well known that XML documents can be regarded as a new type of database, and such data are particularly good for information exchange on the internet. Like relational databases, poorly designed documents may contain too many unnecessary redundancies and these redundancies may contain update anomalies [2, 7, 14, 15]. Data redundancies and anomalies can occur in XML documents if the schema that is DTD (Document Type Definition) [11] or XML Schema [13] is not well defined. In order to avoid these problems, it is very important to have a well defined schema for XML documents. To achieve this aim, a conceptual model Graph Document Type Definition (G-DTD) [16] is proposed to describe XML documents at the schema level. G-DTD has richer syntax and structure which incorporates attribute entity, simple data types, complex element data types, relationship types, hierarchical structure, cardinality, sequence and disjunctions between elements or attributes. The benefit of the G-DTD data model is that, it can be used to capture the syntax and semantics of XML documents in a simple but precise way. Having G-DTD as a tool helps the user to arrange the content of XML documents in order to give a better understanding of DTD structures, improves XML design and assists the normalization process as well. The conceptual model G-DTD is a first layer of an XML document design system which we have formally constructed.

The benefits of having such a formal specification are firstly, to make a precise description of the complete G-DTD model at the conceptual level in order to remove ambiguity that may arise from its graphical representation. Secondly, to make G-DTD itself a modelling notation so that it can be used as the basis for a rigorous tool for XML design and finally, to eliminate inconsistencies in XML design at a schema level. This formal specification is used to describe a fundamental framework of what the system can do and also as an abstraction of a full complete system which can serve as a reliable blueprint for those who want to implement the program later. This formal specification is important before the implementation of the real system is developed, as it allows a designer to understand the big picture of the system and helps to discover error early in the development process.

There is a related work by Anutariya et al. [1], which has proposed a formal data model for an XML database using XML Declarative Description (XDD) theory. However, the most related work using a formal method to present formally a data model for semistructured data called Object Relational Attribute for Semistructured (ORA-SS) is done by Lee et al. [8, 9]. They used different types of formal method languages to present the syntax and semantics of the model. For instance, Lee at al [8] used Z formal language to validate the syntax and semantics of the ORA-SS model. They also validated the model to check the correctness of ORA-SS at both schema and instance levels. Similar to this work, the formalization of ORA-SS using OWL was presented to
improve verification performance. Recently, Lee et al [9] have used a different approach to define a formal specification for ORA-SS using Prototype Verification System (PVS) language. However, to the best of our knowledge, no formal specification has been developed to define an XML document design system. This paper describes the first layer of the system.

The rest of the paper is organized as follows: Section II provides background knowledge on G-DTD notations, structure and operations. Section III presents the Z formal specification of G-DTD. In Section IV, we demonstrate the formal specification of G-DTD operations defined in Section II. We conclude the paper with our future work in Section V.

II. BACKGROUND

DTD is commonly represented as textual representation. In practice, it often causes difficulties when designing even a simple XML document. More importantly, in DTD, the semantic constraints and relationship between the elements in the XML document cannot be represented precisely and clearly. For instance, as shown in Figure 1, the relation between course and student is not defined explicitly. The semantic relation between the elements presents only one-to-many relationships, while other relationships such as many-to-many or many-to-one relationships cannot be defined. However, G-DTD overcomes the above problems by using a graphical notation to visually represent an XML document structure at the schema level. This notations are shown clearly in the example provided in Figure 2. In this way, we believe the user can have a better understanding of XML document structure. Indeed, Mok and Embley [10] make the argument that “the graphical conceptual modelling languages offer one of the best human-oriented ways of describing an application”

Representation of G-DTD is slightly different from the DTD. Firstly, we distinguish explicitly the difference between complex elements, simple element and attribute. We emphasise that a simple element is an element with no child elements, while an attribute is a key or candidate key of a complex element. The reason for this is to make the normalization process easier. Secondly, we present the G-DTD structure as a hierarchical structure of elements which is similar to XML document structure, to provide an accurate picture of the XML document. The advantages of G-DTD over DTD are: it allows users to define explicitly the structure of attribute nodes, simple element nodes and complex element nodes in a hierarchical way and also allows the user to determine the relationship dependency between the nodes.

A. Syntax and Semantics of G-DTD

Some of the notations of G-DTD have been adopted and improved upon from the current data model ORA-SS [5] notations and conventional ER model [4]. G-DTD [15] consists of six basic components:

1. **Complex element node.** A complex element node is used to represent an ‘ELEMENT’ in DTD. The complex element node is illustrated as a labelled rectangular box. This notation is adopted from the ER model [4] which is similar to entity. The label is written in the rectangle as a tuple <name, level>, where name represents the name of the node and level represents the depth of the node in G-DTD.

2. **Simple element node.** A simple element node is used to represent an ‘ELEMENT’ associated with #PCDATA or #CDATA. It is illustrated as a labelled rounded rectangular box with the form <name, level, type> where name is the name of the simple element, level is the depth of the node in the G-DTD and type represents PCDATA or CDATA or string ‘S’. All simple element nodes are assumed to be mandatory and single valued, unless the node contains the symbol ‘?’ which signifies it is single valued and optional, or + which signifies that it is multi-valued and required, or an * which shows that it is optional and multi-valued. This notation is similar to ORA-SS [6]. The symbol is written in front of the tuple <name, level, type> to differentiate among them accordingly.

3. **Attribute node.** An attribute node is used to represent an attribute defined in ATTLIST. The attribute node is an identifier for a complex element node. It is represented as an ID which is unique and mandatory among the instances of complex elements. Attributes can be classified as single attributes and composite attributes. A single identifier attribute has an atomic value and composite attributes have more than one identifier attributes. A single identifier attribute is represented as an oval and a composite attribute as a double oval.

4. **Set relationship type.** Three types of relationship are used in G-DTD: Hierarchical link, part_of link and has_a link. The Hierarchical link is a relationship between complex element nodes. This link shows the relationship between parent node to child node or ancestor node to descendant node. For Hierarchical link, a relationship dependency, which is indicated by the connectivity between complex element occurrences, is important. Basic constructs for connectivity are: one-to-one (unary or binary relationship), one-to-many (unary or binary relationship), many-to-one and many-to-many (unary or binary relationship). All these types of relationship are indicated by directional arrows. The notation is presented as (name, d, cp, cc) where name represents the name of the relationship, d is the degree of relationship, cp and cc are cardinality constraints for parent and child respectively. This notation is similar to ORA-SS [6]. The degree can be two, three or n-ary. The cardinality of cp and cc in a relationship is represented as 2 tuple (min: max). The constraint (0:N), (0:1)and (1:N) is represented as the
operators *, ?, and + respectively, except the cardinality constraint (1:1) is presented as 1. For instance, the diagram in Figure 2 illustrates a binary Hierarchical link between complex element student and complex element courses, where a student can take zero or many courses while many courses can be taken by zero or many students. Part_of_link is a relationship between a complex element node and an attribute node. It is illustrated as a bold double arrow. Has_a_link is a relationship between complex element node and a simple element node. It is illustrated as a single double arrow.

(5) Semantic constraint between set relationships. There are two types of set relationships: First, sequence between a set of child element nodes. We emphasize in our notation that the attribute node(s) must be located in the first position in the sequence. To express such ordering in a G-DTD, we draw a directed upwardly curving arrow labelled with [sequence] across all the set of relationships involved. Second, is disjunction between the set of sibling nodes. To illustrate this, we draw a line labelled with [XOR] across all the set of relationships involved.

(6) Root node. A root node is used to represent DOCTYPE. Its notation is similar to complex element notation, as it is a special case of a complex element node and its level is always zero.

Figure 2 shows a G-DTD describing the structure of an XML document corresponding to the DTD in Figure 1. The root node Department has a binary hierarchical link with the complex element node course. The semantic relationship between them reveals that the Department can have one-to-many courses at one time. The complex element course has a sequence of attribute cno, simple element node title and complex element node student.

```xml
<!DOCTYPE department[
<!ELEMENT department(courses*)>
<!ELEMENT course(title, student*)>
<!ATTLIST course cno ID #REQUIRED>
<!ELEMENT title (#PCDATA)> 
<!ELEMENT student(frame?, lecturer)> 
<!ATTLIST student sno ID #REQUIRED>
<!ELEMENT frame (#PCDATA)>
<!ELEMENT lecturer (name)> 
<!ATTLIST lecturer tno ID #REQUIRED>
<!ELEMENT name (#PCDATA)>
]> 
```

Figure 1. A DTD for the university database

The part-of link attribute is a mandatory relationship where the attribute node cno is required and unique for every course in the XML document. The simple element node title is part-of the complex element courses. One course can be taken by many students while the complex element student consists of a sequence of attribute node sno, simple elements fname, lname and complex element lecturer. Attribute node sno is required for the complex element student. Complex element node student requires only one of its subelements, either fname or lname, to appear in the XML document while the simple element lname is optional. The semantic relationship between course, student and lecturer is indicated as a ternary relationship since each student is assigned to a lecturer who is teaching the course.

As shown in Figure 2, the semantic relationships between the complex element nodes have been added at the hierarchical link to present more semantics at the schema level. The reason we add this type of semantics is to make the relationship between the nodes more explicit, which will help during the normalization process.

![G-DTD Diagram](image-url)
B. G-DTD Operations

The operations of the G-DTD model describe the dynamic properties of the model. G-DTD model operations are classified into five main parts: Query Operations, Insert Operations, Delete Operations, Searching Operations, and Update operations. An operation to determine the root and leaves of the G-DTD is also required. Later, these operations will be used in normalizing the G-DTD into normal forms. In the following description, we will conceptually discuss the semantic connection of these operations according to this classification.

(1) Query Operations

Query operations allow the user to query the node types and information, related nodes and links information defined in G-DTD.

(a) Query a Node Type and Information

The operations of querying node types allow the user to query different types of node stored in G-DTD such as complex element, simple element or attribute nodes. The user can also query information of a particular node, such as name, level and node type. If the queried node does not exist, an error message is given.

(b) Query a Related Node

Since the structure of G-DTD is like a tree structure, the query operations allow the user to query the related node that links to a particular node using a path through an existing link such as a Hierarchical, Part_of or Has_A link. For instance, the user can detect the parent of a complex element node by using the hierarchical link between two complex element nodes. Another example, the simple element for a particular complex element node can be determined through the has-a link.

(c) Query a Hierarchical Link

Hierarchical links are the most important links in G-DTD. This operation allows the user to query the instance of a hierarchical link, such as name of link, degree of relations and parent and child constraint.

(2) Insert Operations

Insert operations allow the user to add new nodes to the G-DTD. When a new node is being inserted in the G-DTD model, the following situations are possible:

- A part-of link is built between the created complex element node and an attribute node

To ensure the new node is not redundant with any node in the given G-DTD, it must be checked whether the node already exists. Then the proper location of the new node needs to be determined before it can be inserted into the G-DTD. More importantly, it must satisfy the data integrity constraint of the given G-DTD.

(a) Inserting a Node

In this case a new node is inserted into the G-DTD. Whether the new node is a complex element, simple element or attribute node, the properties of the inserted node such as ID, level and types are inserted and stored together in the G-DTD. The operation implies that when the node is inserted, related nodes such as parent node or child node should be reported to the user since the structure of the G-DTD is changed. If the newly inserted node is a complex element node, the position of the new complex element node is based on the rules provided in the normalization procedure [17]. In such a situation, a hierarchical link is created with its parent node. In this case, the parent node may be a root node or another complex element node based on the normalization rules provided. However if the created node is a simple element or an attribute node, a Part_of link or Has_A link is built between it and the parent node, which is a complex element node.

(b) Inserting an Instance of a Hierarchical Link

Inserting an instance of a hierarchical link means that the semantic relation between two complex element nodes has to be created. The user needs to know the semantic relationships before he/she can insert them to the G-DTD. The user can make links and insert the corresponding link information such as name, degree, parent constraint and child constraint. In contrast, for a Part_of link or Has-A link, the user is not required to put any instance for the links.

(3) Delete Operations

Delete operations result in the corresponding data being removed from the G-DTD. Since the structure defined in the G-DTD is a tree structure, deleting will affect the location of the existing nodes in the G-DTD, especially the parent node and child node. The delete operation in G-DTD must satisfy the conditions and constraints given in the normalization rules [17]. In the following, we will discuss the different situations of delete operations in the G-DTD.

(a) Deleting a Complex Element Node

Deleting a complex element node is a complex deletion process in G-DTD. This is because every complex element node is related to its parent node and child node. Before the deletion process of a complex
element node is started, it is important for the user to find its related nodes such as its parent node and child nodes. Eventually, by deleting a complex element node, its attribute and simple element nodes with the relevant, Part_of and Has_A links are automatically deleted as well. Then, new links are built up with its new parent node and child node.

(b) Deleting a Hierarchical Link type and its Instance

According to the hierarchical link type definition, each instance of a hierarchical link type represents a semantic relationship between two complex element nodes. When such an instance is deleted, the specific relationship between the two nodes has no further semantic link between them.

(4) Update Operations

Update operations change the location of the current node. A complex element node or simple element node can be moved around from one location to another. In the process of moving a node, all the related nodes including complex element nodes and simple element nodes should be notified if the moving node has a relationship with them. The only case we consider here is moving a complex element node. It may be necessary to move a complex element node up to another level when there exists dependency between an attribute node and simple element node of a complex element node. In this situation, it is not necessary to create a new element node but rather to restructure the G-DTD by moving up the complex element node at level n \((n_0)\) to level n-1 \((n_{n-1})\) along with its corresponding children.

(5) Determine the root node and last node

This operation will determine the root node and last node (last level) in the G-DTD. The last node may be a simple element node or attribute node. These operations are very important because in order to avoid duplication, we need to move the corresponding node to a position as close as possible to the root node.

III. THE SPECIFICATION OF G-DTD

In this paper we provide a formal specification of the G-DTD which represents a formal, concise and readable definition of the G-DTD and its operations. The specification can be used as the basis for implementation, as well as a framework for further XML document design. We choose the language Z [11] to formalise our model for a number of reasons. First, the language is based upon primitive mathematical notation such as set theory and first order predicate logic, making it accessible to researchers from variety of different backgrounds. Second, it is expressive enough to allow consistent, formal and unified representation of a system and its associated operations. Third, it is model oriented [3]. A model-oriented specification language seems more appropriate to specify an XML design model and it is easier to understand. Finally, in particular, we have found that Z is an established language, widely accepted and appropriate for building formal frameworks [9]. A specification written in Z is a mixture of formal mathematical statements and informal explanatory text. Both have their importance: the formal part gives a precise definition of the system being specified, while the informal text makes the specification more comprehensive and readable, linking the abstract definition of the system to the real world. In this paper we present only some basic components and operations, due mainly to space limitations; other results will be published in a forthcoming paper.

A. Basic types

We use the basic types \([ID, \ Element\_Name, \ Attribute\_Name, \ Relation\_Name]\) as a given set which will be used in the later schema definition. ID represents each nodes identifier, which is unique; both \(\ Element\_Name\) and \(\ Attribute\_Name\) are used to represent the set of all possible XML element nodes and attribute nodes respectively. \(\ Relation\_Name\) is a set for relationship names.

B. The Data Structure of G-DTD

As described in Section II(A), we captured the characteristics of each type of node such as simple element, complex element and attribute nodes using the following schema type. There is no constraint we need to add in each of the declarations.

(1) Simple Element Node

The type definition for a simple element is defined as follows:

\[
\text{Simple\_Element\_Type} ::= \text{singlevalue}| \text{multivalue}\mid \text{op\_singlevalue}\mid \text{op\_multivalue}
\]

\[
\text{Simple\_Element\_Node} ::= \text{Identity}\mid \text{name}\mid \text{level}\mid \text{elementType}\::\text{Simple\_Element\_Type}
\]

(2) Attribute Node

The AttributeNode schema captures the properties of an attribute node as follows:

\[
\text{Attribute\_Type} ::= \text{composite}\mid \text{required}\mid \text{reference}
\]

\[
\text{Attribute\_Node} ::= \text{Identity}\mid \text{name}\mid \text{level}\mid \text{AttType}\::\text{Attribute\_Type}
\]

(3) Complex Element Node

The ComplexElementNode schema represents the properties of a complex element node with its identity, name and level.
(4) Parent for Complex Element Node, Simple Element Node and Attribute Node

Because the structure of the G-DTD is a tree structure, it is important to define a parent for each complex element node, simple element node and attribute node to describe precisely the relationship between them. The functions parent_ce, parent_se and parent_att are defined using the axiomatic function as a total function because every complex element node, simple element node and attribute node must have its own parent node and no node can have more than one parent.

\[
\begin{align*}
\text{parent_ce: ComplexElementNode} & \rightarrow \text{ComplexElementNode} \\
\text{parent_se: SimpleElementNode} & \rightarrow \text{ComplexElementNode} \\
\text{parent_att: AttributeNode} & \rightarrow \text{ComplexElementNode}
\end{align*}
\]

In the state invariant, it is stated that complex element ce1 \(\mapsto\) ce2 \(\in\) parent_ce means that ce2 is the parent of ce1 if and only if ce1 is not the same as ce2 and the level position of ce2 must always be less than the level position of ce1 by one level difference only. The same meaning is applied for the second and third predicates associated with the parent for a simple element node and parent for an attribute node, respectively.

(5) Relationship

We define three types of relationship which are Hierarchical_Link, Part_of_Link, and Has_A_Link using the following schemas.

(a) Hierarchical_Link

The Hierarchical_Link schema consists of a relation hierarchical_link which is used to define a homogeneous relation between complex element nodes. The first and second predicates of the schema state that an ordered pair of complex element nodes ce1 \(\mapsto\) ce2 is an element of hierarchical_link if and only if ce2 is an immediate parent of ce1 or ce2 is a hierarchical parent of ce1, ce1 \(\mapsto\) ce2 \(\in\) hierarchical_link *, that is to say, it is a transitive closure relation. The third predicate of the schema defines that the child complex element should not be the same set as the parent complex element node and finally the relation must be cycle free, which means no complex element node is mapped to itself. This is defined using transitive closure to capture the idea of some complex element nodes (homogeneous binary relation) can be directly reached in the same link. The relation hierarchical_link is known as a homogeneous relation [4] since the complex elements are from the same set. One of the benefit of this relation is that it can be composed among such links themselves. Thus, we can form the relation hierarchical_link; hierarchical_link. This can also be written as hierarchical_link^2. The hierarchical_link can be repeated as many times as desired. The constraint relationship on the hierarchical_link must be a positive number. The properties of the schema also consist of name, degree of relationship, parent cardinality and child cardinality constraints.

\[
\begin{align*}
\text{Hierarchical_Link} \\
\text{Hierarchical_link: ComplexElementNode} & \rightarrow \text{ComplexElementNode} \\
\text{degree: } &N
\end{align*}
\]

(b) Part_of_Link

The Part_of_link is a binary relationship rather than n-ary relationship. It consists of Attribute_key function and Composite_key relation. The Attribute_key function is a total and injective type because each complex element node has a unique attribute node. The Composite_key relation is a relation between a complex element and attributes. In the first predicate, ce \(\mapsto\) att \(\in\) Attribute_key if and only if the attribute type is required. The second predicate states that, ce \(\mapsto\) att \(\in\) Composite_key if and only if the attribute type is composite. The last predicate indicates that the domain for the Attribute_key function and Composite_key relation is a member of a complex element node.

(c) Has_A_Link

The schema Has_A consists of a has_a relation which describes that a complex element node has a relation with a simple element node where a simple element can be a single value, multivalue, optional single value or optional multivalue and must have a complex element node as a parent.
C. The State Space of Schema G-DTD

To finally organize the structure of the G-DTD, all the above-defined node types and relationship types are used in the schemaGDTD definition.

The SchemaGDTD consists of seven variables which include a root node type, set of ComplexElementNode, set of SimpleElementNode, set of AttributeNode and set of relation HierarchicalLink, Has_A and Part_of types. The first predicate of the SchemaGDTD states that there must exist one root node. The second, third and fourth predicates indicate that at any point in time, each complex element node, simple element node and attribute node must have a unique name. The last four predicates ensure that all types of nodes and relationships defined exist in SchemaGDTD.

\[
\text{SchemaGDTD} = \text{root:ComplexElementNode} \cup \text{Cnodes:ComplexElementNode} \cup \text{Snodes:SimpleElementNode} \cup \text{Attnodes:AttributeNode} \cup \text{HierarchicalLink:HierarchicalLink} \cup \text{HasA:Has_A} \cup \text{Partof:Part_of}
\]

D. Initial State of Schema G-DTD

Before any operation can be performed on the model, we must define the initial state of the G-DTD. In our case, the initial state of the G-DTD refers to the situation in which there are no elements existing in the schema. This schema describes the InitialG-DTD in which the sets of simple element nodes, complex element nodes and attribute nodes are empty; in consequence, the HierarchicalLink, HasA and Partof relations are empty too. This is characterized by the following schema definition:

\[
\text{InitialG-DTD} = \text{\Delta SchemaGDTD} = \text{Snodes} = \emptyset \cup \text{Cnodes} = \emptyset \cup \text{Attnodes} = \emptyset \cup \text{Partof} = \emptyset \cup \text{HasA} = \emptyset \cup \text{HierarchicalLink} = \emptyset
\]

IV. OPERATIONS SPECIFICATION IN G-DTD

The operations defined in schema G-DTD describe the behaviour or state change of the G-DTD during editing and manipulating nodes. We present some of the operations which are query operations, create, insert and delete operations. However, before we present these operations we must first define the following functions.

1. Create Complex Element Node

2. Create Attribute Node

3. Create Has_a link

The first predicate of the function assigns an instance of a new complex element node. The second predicate gives a pre-condition for the success of the operation. The new complex element to be added must not already be one of the members of complex element nodes in G-DTD. This is because only one unique complex element is allowed in the G-DTD schema. If this condition is satisfied, the new complex element node is added to the set of complex element nodes.

The description of the Create_AttributeNode function is similar to the Create_ComplexElementNode function.

Create_Has_a_Link is a function to create a new Has_a link between a complex element node and a simple element node. The first predicate of the function maps both of the given complex element node and simple element node and assigns between them a new has link. Then the new has link is added to the set of new has links in SchemaGDTD.
(4) Create Hierarchical link

Create_Hierarchical_Link : (ComplexElementNode x ComplexElementNode) -> HierarchicalLink

∧ ce1,ce2: ComplexElementNode; schema: SchemaGDTD •
(∃new_HierarchicalLink, newlink: HierarchicalLink; level: N1; pc, ce : N0;)
new_HierarchicalLink = newlink •
(ce1⇒ce2 ∈ newlink, hierarchical_link ⇒
∧ ce1 = ce2 ⇒ newlink, hierarchical_link •
∧ A ce2 = parent (ce1)
∧ name (newlink) = newname
∧ A degree (newlink) = level
∧ A parentconstraint (newlink, hierarchical_link) = pc
∧ A schema_Hierarchical_Link =
schema_Hierarchical_Link U
{ new_HierarchicalLink }
⇒ Create_Hierarchical_Link (ce1,ce2) = new_HierarchicalLink

(5) Create Partof link

The function Create_partof_Link is used to create part-of links between complex element nodes and attribute nodes. The argument of this function is a relation between a complex element node and attribute node and return a part of link. The new part_of link can be either Attributekey or Compositekey and the parent of the attribute node must be a complex element node. Finally, a new part_of link is added to the set of part_of links in SchemaGDTD.

create_Partof_Link : (ComplexElementNode x AttributeNode) -> part_of
∧ ccc: ComplexElementNode; att: AttributeNode, new_partoflink , partoflink: part_of; schema: SchemaGDTD •
∧ schema : SchemaGDTD •
new_partoflink = partoflink
∧ ce=att ∈ partoflink, AttributeKey ⇒
∧ getattType = required ∧ parent_att(att) = ce
∧ ce = att ∈ partoflink
∧ A getattType = composite ∧ parent_att(att) = ce
∧ A schema.Partof = schema.Partof U (new_partoflink )
⇒ create_Partof_Link (ce, att) = new_partoflink

A. Query Operations

Before manipulating the structure of any complex element node in the G-DTD, we should be aware of its related nodes. Since the structure of the G-DTD is like a tree structure, a child or descendants and parent or ancestor of a given complex element node needs to be queried in some cases. The status of a queried node is defined using a set of messages. It is defined by enumeration type

Report:: = Existence | Nonexistence | Inserted | Created

Based on this set, we define the following schema Success to output a confirmatory message that the operation being performed has been successfully completed.

 Success
 report! : Report
report = Existence

The following Get_AttributeKey shows how to get an attribute key of complex element node using the part_of link

Get_AttributeKey

∧ SchemaGDTD
∧ ce : ComplexElementNode
∧ attkey : AttributeNode
∧ attkey = part_of_AttributeKey (ce)

Get_SimpleElement schema captures how to get a simple element node by using has_a link

Get_SimpleElement

∧ SchemaGDTD
∧ ce: ComplexElementNode
∧ se ∈ SimpleElementNode
∧ has_link : HasA •
∧ se1 = has_link, hasa \{ce\}

Each operation can only go wrong if the complex element ce is not in SchemaGDTD. This case is captured by means of the schema UnknownNode.

UnknownNode

∧ SchemaGDTD
∧ ce : ComplexElementNode
∧ report! : Report
∧ ce ∈ dom has_link, hasa •
∧ ce ∈ dom HierarchicalLink, hierarchical_link
∧ report = Nonexistence

Based on the schema definition above, we can finally define the following schemas, which describe the state in which a simple element node or attribute node has been successfully queried.

Do_Query_AttributeKey = Get_AttributeKey A Success V UnknownNode
Do_Query_SimpleElementNode = Get_SimpleElementNode A Success V UnknownNode

The following schema is used to capture the query operation for a complex element node. This schema means that the existing complex element node whose name is equal to the input name is found.

Get_ComplexElementNode

∧ SchemaGDTD
∧ ce : ComplexElementNode
∧ ce_name : Element_Name
∧ ce : ComplexElementNode
∧ found_ce : Element_Name = ComplexElementNode
∧ ce : ComplexElementNode •
∧ ce_name = ce_name ⇒ found_ce ce_name = ce

Do_Query_ComplexElementNode = Get_ComplexElementNode A Success V UnknownNode

A query about the ancestor or descendants of complex element node can be made by using a Hierarchical_Link. We achieve this by forming the transitive closure of Hierarchical_Link

Ancestors

∧ SchemaGDTD
∧ ce : ComplexElementNode
∧ ancestor : ComplexElementNode
∧ ancestor ce! = (hl. HierarchicalLink) * \{ce\}

Do_Query_AncestorsNode = Ancestors A Success

Descendants

∧ SchemaGDTD
∧ ce : ComplexElementNode
∧ descendant : ComplexElementNode
∧ descendant ce! = (hl. HierarchicalLink) * \{ce\}

Do_Query_Descendants = Descendants A Success

B. Insert Operation

Insert_NewComplexElement_Node schema is used to insert a new complex element into G-DTD. In the
signature of the schema, the declaration $\Delta\text{SchemaGDTD}$ alerts the user to the fact that the schema is describing a state change. The functions $\text{Create\_New\_ComplexElement}$ and $\text{Create\_Hierarchical\_Link}$ are used to create a new node and create a new link respectively. Before the node can be inserted, a pre-condition is given to check whether its exists already. The new complex element to be inserted must not already be one in the G-DTD. This is because only one unique complex element is allowed in the G-DTD schema. If this condition is satisfied, the new complex element node is inserted and a hierarchical link is created between the new node and its parent node. When the operation is successful, the output will take a value inserted.

$$\text{Insert\_New\_ComplexElement\_Node}$$

$\Delta\text{SchemaGDTD}$

level? : N
newname? : Element_Name
newid? : ID

newnode = ComplexElementNode ; newlink = HierarchicalLink
newnode = Create\_New\_ComplexElement(newid?, newname?, level?) ∧ newlink = Create\_Hierarchical\_Link(newnode, parent(ce(newnode)))

The schema success just outputs a confirmatory message that the operation being performed has been successfully completed.

$$\text{Success}$$

rep! = Report

rep! = Inserted

To capture the condition where the simple element node is already a member of G-DTD, the following schema is used:

$$\text{AlreadyExisted}$$

$\Delta\text{SchemaGDTD}$

se? : SimpleElementNode
found_se? = SimpleElementNode_report! = Report

se? = SimpleElementNode ∧ se.name = se? !⇒ found_se? = se! ∧ Report! = Existed

To perform $\text{Do\_Insert\_New\_ComplexElement\_Node}$ operation the following is used.

$$\text{Do\_Insert\_New\_ComplexElement\_Node} \equiv \text{Insert\_New\_ComplexElement\_Node} \land \text{Success} \lor \text{AlreadyExisted}$$

C. Delete Operation

The operation to delete a simple element node from the G-DTD is specified by the following schema:

$$\text{Delete\_Simple\_Elements\_Node}$$

$\Delta\text{SchemaGDTD}$

Get\_Simple\_Elements\_Node

se? : SimpleElementNode

se? = SimpleElementNode ∧ parent(se? , link = hasa | parent_se? , link = link(se? )) ∧ delete_partofLink(parent_se? , link, schema )

newnodes = newnodes \{ se? 

Before the node can be deleted, it must be checked that the given node is a member of simple element nodes in the G-DTD and the parent of the simple element node needs to be determined. The node can be deleted from the G-DTD if the input node is present in the G-DTD. If this pre-condition is not satisfied, then this will be captured by the following schema:

$$\text{UnknownNode}$$

$$\Delta\text{SchemaGDTD}$$

se? : SimpleElementNode

report! = Report

se? ∈ Snodes

report! = Nonexistence

The complete specification of the operation to delete a simple element node from $\text{SchemaGDTD}$ is given by the schema:

$$\text{Do\_Delete\_Simple\_Element\_Node} \equiv \text{Delete\_Simple\_Element} \land \text{success} \lor \text{UnknownNode}$$

V. CONCLUSION

We have presented a formal specification of a G-DTD model using Z notation style which gives precise, mathematical meaning to basic conceptual structures. The formalization of the G-DTD model is required for a deeper understanding of modelled syntax, structure, and semantics of model properties. The use of formal specification techniques contributes to the clarity and conciseness of the model, and enables formal derivation of model properties to be performed easily. Obviously, this paper has reported only the beginning of formal development of an XML document design model, since it includes just a description of the G-DTD model structure and its basic operation. Currently we have constructed a complete formal specification for an XML document design model using G-DTD by applying those functions and schemas (defined in Sections III and IV). This specification includes finding of various functional dependencies, checking the G-DTD normal forms and normalization procedure operation. However, these results will be the subject of another paper.

REFERENCES


