ZML: Z Formal Specifications Using XML

T. Scott Ankrum
University of Maryland University College
Scott_Ankrum@acm.org

Abdelghani Bellaachia
George Washington University
bella@seas.gwu.edu

January 16, 2003

Abstract

This paper describes a project, the purpose of which was to determine the feasibility of using XML as a vehicle for carrying the contextual meaning of a Z formal specification. This paper describes a definition of the popular Z (pronounced zěd), formal specification language, in terms of an eXtensible Markup Language (XML) Data Type Definition (DTD). Using the Sun JAXB tool, we wrote a Java program that supports building and processing a tree structure based on the DTD. Using that DTD, a small Z specification was rendered in XML. We then present a case study Java program written to demonstrate the ability to examine and process the Z specification as an internal tree structure. Finally, we describe some limitations encountered in the JAXB tool and suggestions for improving it. With the feasibility proven, we could then go on to use the XML vehicle to pass the Z specification to a theorem prover, a code generator, or other, as yet unforeseen tools.

Z is a formal specification language, used for specifying requirements or designs, and to which formal proofs of correctness and completeness can be applied. XML allows us to define a Z schema, so it can be used with other software tools to contextually render the schema. We describe a complete Data Type Definition (DTD), denoted ***ZML***, for the general Z schema format. While pursuing our primary purpose, we have found some limitations in JAXB, which we describe in this paper.

1. Introduction

Many of the problems that are identified through vigorous software testing stem from incorrect and/or ambiguous software requirement specifications. This has led to the introduction of formal methods; mathematically based models of requirement and design specifications of software systems. These are used to ensure system correctness for safety-critical systems, security-critical systems and others. Z (Spivey, 2001), pronounced zěd, a mathematically based formal specification language, has been a very popular formal language for writing formal specifications for software systems. Specifications written in Z can be formally proven consistent and correct. Z is only one of several formal specification languages that has been used successfully one several software development projects. The use of a formal specification language can reduce the number of software defects and improve productivity, because proofs of the specification can avoid defects in the code rather than having tests remove defects later.

We use Extensible Markup Language (XML) (W3C, 2002), which is a language for data interchange to represent and store Z specifications. XML allows us to represent a Z specification that can be easily rendered and integrated into other software components. We call the resulting definition *ZML*, for Z Markup Language. By "contextually render" we mean that XML allows us to preserve the meaning of each part of the Z specification as it relates to the parts around it. Once the general form of a Z schema is defined with a DTD, any Z schema can be read contextually. This opens many possibilities for transforming Z specifications into different forms and for different uses. A detailed design specification, for example, could be interpreted for the purpose of generating program code. A properly tagged schema can be rendered in different ways by different tools for display or print in various formats or for uses that may not yet be foreseen.

In this paper, we describe a complete Data Type Definition (DTD), of ***ZML***, for the general Z schema format. We developed a software component, using the JAXB tool (Sun Microsystems, 2002), , that supports the rendering of the ZML specifications.. The purpose of this work is to prepare the way for working with Z specifications for many different purposes, including generating executable code from design specifications and theorem proving. While pursuing this primary purpose, we have found some limitations in JAXB, which we describe below.

The remainder of this paper includes the following. Section 2 describes the project upon which this paper is based, including the tools used, project goals and related work. Section 3 provides an overview of XML and Sun's JAXB tool. Section 4 explains the structure of the Z language and how it is encoded in the DTD. Section 5 describes some changes made to the original DTD necessitated by JAXB limitations. Section 6 introduces the JAXB binding scheme, how it is used and why. Section 7 describes the case study used to exercise the ZML work presented in this paper. Section 8 lists recommendations for improving JAXB based on problems encountered in our work. Section 9 is this paper's conclusion.

2. Outline of the Project

2.1. Project Goals

This project is a continuation of work begun by Ankrum and Appiah (2001). That project used the eXstensible Stylesheet Language (XSL) to render Z XML documents on a web page. In that first project, we rendered the XML documents into a web page that crudely resembled the original Z specifications. We found that XSL is not powerful enough by itself to allow for a faithful recreation

of the Z schema format. Specifically, when an XML document carries information as both elements and attributes, it is very difficult for XSL to intermix that information in the desired order. The natural inclination, when using XSL, is for all of the attribute values to be displayed ahead of the value of the element itself. Rendering any other specific ordering of element and attribute values is problematic. The primary goal of this project was to identify a mix of tools that would allow a complex XML file, holding a Z specification, to be analyzed and/or rendered with the most flexibility and the least effort.

2.2. Tools used

To bring more power and flexibility to bear on the task of rendering the XML documents, we decided to write program code in the Java language. The most obvious tools to support rendering using Java would be either the Document Object Model (DOM) or Simple API for XML (SAX) interface. The trouble with these tools is that they present the tree structure based solely upon the XML document being read, instead of on the DTD. Because the DTD defines the general form of the XML document, a tree based upon the DTD will present a clearer picture for traversing the complex structures required for a Z schema.

Sun Microsystems' Java Architecture for XML Binding (JAXB) seemed to offer an ideal interface for our purposes. It generates Java program code, with a Java class for each XML element. The developer then writes code to invoke these classes, giving the developer a view of the entire XML file as an internal tree structure. See the section Overview of JAXB, below.

The other significant tool used is IBM's Xeena, still in prerelease test. Xeena is an XML editor and validator. Given a DTD while editing an XML file, it can point to where the XML file is invalid, relative to the DTD, as well as when the XML file is not well formed. A well-formed XML file is one that adheres to all of the general rules of XML tags and documents. A valid XML file is well formed and also adheres to all of the rules defined in a given DTD.

2.3. Related Work

Some work has been done to capture Z in XML. At the National University of Singapore, a graduate student team (Sun et al, 2001) concentrated on translating Object-Z, an object oriented derivative of Z, to and from Unified Modeling Language (UML). That work differs from this in several important respects. First, it encodes Object-Z, which is mostly an extension of the Z language. Second, it is concerned only with translation of Object-Z to and from UML, whereas we have more open ended goals. Finally, they have spent significant effort on properly displaying the Z symbols. We have concentrated on encoding the meaning of a Z specification, leaving display concerns to a later output tool.

The Community Z Tools Initiative (CZT) (Martin, 2001) lays out a plan for a whole set of open source Z tools. Based on the number of updates and the latest date on the web site, the activity of the community seems very low. The project is based on a DTD specification of the Z language that was begun by Ian Toyn of the University of York in January 2002. Most importantly, work on CZT's DTD was begun after the earlier project that defined our DTD had finished. Because that DTD was developed later than ours and largely in parallel with our later project, we have not yet investigated the extent to which the two projects overlap.

The project for this paper was based on earlier work by Ankrum and Appiah (2001). In that project an initial DTD was specified and demonstrated to be correct through the use of an XSL stylesheet to display an example Z schema. That initial DTD has been retained, with only a few changes, for this paper's project.

3. Overview of XML and JAXB

XML is a currently popular language for creating self-describing documents. It is similar to Hypertext Markup Language (HTML) and Standard Generalized Markup Language (SGML) in that human readable text tags describe each component or element of the document. It differs from those languages in that the author of a document can design a tag structure to suite the needs of the document. A file called a DTD defines all of the tags and how the text elements can be structured to form a document. The DTD shows, for each text element, what combinations of other elements and raw text it can contain. These definitions can be nested in various complex combinations. The DTD used for the project is explained later in this paper.

JAXB builds Java program code based upon the DTD and what they call a binding schema, which supplements the DTD with additional information necessary to build the Java program. We explain the binding schema in a later section of the paper. JAXB is a program that takes two files as input: a DTD and a Binding Schema. Its output is Java program code that forms the shell of a program that must be completed by the developer. The shell consists of Java classes that each represent an element in the DTD, plus some Java methods that allow an invoking program to build an internal tree representing an XML document based on the DTD, and to navigate through that tree. The internal tree contains a node for each element in the XML document that is read in by the program. In general, JAXB generates a class for each element that contains mixed or element content. The value of an element that only contains character data is held in an instance variable of its parent element. The values of attributes are also held in instance variables.

A simple DTD needs only what JAXB calls the minimum binding schema, which only designates the top-level element as a root element. In such a simple DTD, each element can only contain either character data or a sequence of elements. Any element that contains a choice of elements or a combination of sequences and choices requires entries in the binding schema to guide JAXB in generating useable Java classes.

4. Explanation of the Z DTD

A diagram of the general form of a Z specification as a tree structure is in Figure 1. That is, the diagram represents the general for of any Z specification, and it was used as the basis for creating the DTD. In the diagram, each box represents an element. The box at the top of the tree is the root element, as designated in the JAXB binding schema. Each box with a double boundary represents an element with mixed or element content, or at least with attributes. JAXB will represent each of these elements by a Java class. Those boxes with a single boundary represent elements with only character data content and no attributes. We expect JAXB to create an instance variable in the parent class for each of these elements. The names not in boxes represent attributes of the elements above them, and they will also be instance variables. The element names convey some of the same information. Elements whose names begin with a lower case letter contain only character data, and we expect them to resolve to instance variables. Elements whose names begin with a capital letter are complex, and we expect them to resolve to Java classes.

We can see in the diagram that the root element, ZSpec consists of a typeList, a Schema and an explain element. Each typeList element is merely the name of some type that will be used in a declaration, and there can be any number of these. Schema is exactly one Z schema and explain is just a textual comment. The Name of a schema might be decorated with delta (Δ) or xi (Ξ), and that is an attribute. The same is true of ExSchema, which is a reference to an external schema. There

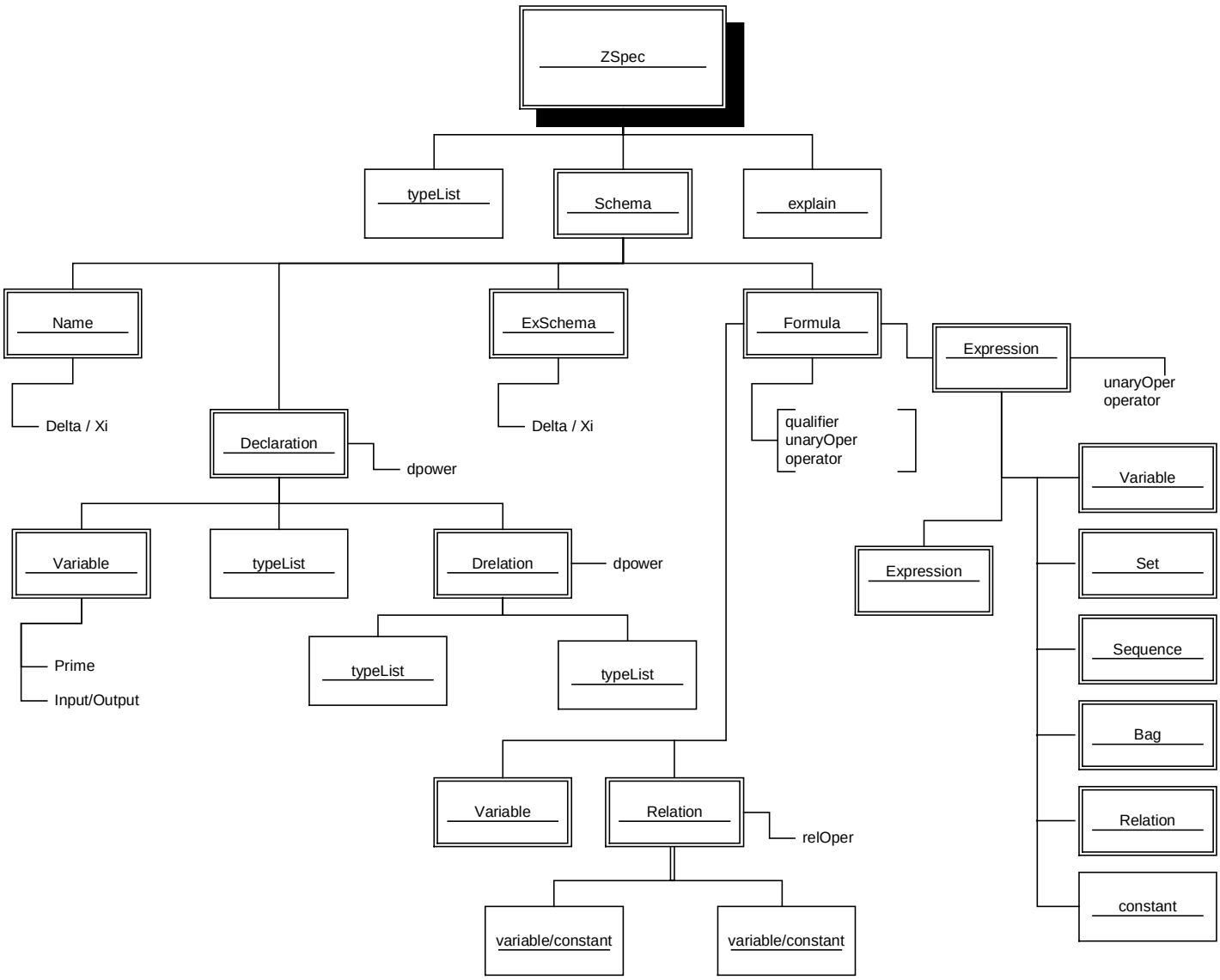

Figure 1. Z Specification Tree Diagram

can be any number of Declaration elements, one for each line in the upper portion of the schema. There is also one Formula element for each formula in the lower portion of the schema. We see that a Declaration consists of one or more Variables declared either as one of the typeList types or as a relation in the Drelation element. Each Variable might be decorated with either or both of two attributes. One is the prime (ɜ) designation (changed), and the other can designate either input or output. The dpower attribute of Declaration can specify a power-set of some type. The Drelation element is used when a variable is declared as a relation between two types, designated with typeList. The attribute relOper holds the relational operator. A Formula is simple at the highest level, but can get complex. It consists of a Variable or Relation on the left and an Expression on the

right. Attributes hold the operator in between, an optional unary operator and a possible universal or existential qualifier. A Relation is similar to the Drelation element, but it is between either two variables or two constants. On the other side of Formula, an Expression can be recursive. It consists of an optional expression and one of the six elements shown in the diagram, separated by an operator, plus a unary operator for the final element. The simplest expression does not contain another expression, so it consists of only one element with an optional unary operator.

5. Deviations in DTD to Address JAXB Limitations

Several limitations of JAXB require that the DTD deviate somewhat from the diagram and the original DTD. The purpose of using a tool like JAXB is to reduce the effort required to design and code the JAVA program that processes the XML representation of the Z specification. Complex areas of the DTD that JAXB does not aid in processing require some approach that helps JAXB to address that complexity. Appendix A contains the DTD used to define the general form of a Z specification. Originally, the DTD followed the diagram precisely. That form was adjusted in small steps to account for the specific needs and shortcomings of JAXB. One minor deviation was to remove all hyphens from element names, because the generated Java class and variable names cannot contain hyphens.

5.1. Nested Repetition

The first significant problem encountered was one of nesting within an element. A Z specification consists of one or more groups of typeList, Schema and explain elements, and each group can include any number of typeList elements. Thus, the original ZSpec element definition looked like this:

<!ELEMENT Z-Spec ((typeList*, explain?, Schema)*)>

This is more than a simple sequence and proved too complex for JAXB. It generated Java classes for the simple elements typeList and explain, which added unnecessary complexity. The added complexity would be in the developed Java code that invokes the JAXB generated code. To circumvent the problem, an intermediate element, Spec was introduced. The result can be seen as the first two elements of the DTD in Appendix A. By introducing the Spec element, we avoid a single definition with nesting, but we require additional elements to be coded in each XML document.

5.2. Attribute of an Empty Element

In our original work, most of the operators, as in relations and formulas, as well as qualifiers and the power-set indicator, were each held in an empty element as a single attribute. This was an obvious and satisfactory way to carry such a value. However, when working with JAXB, we saw that any element with an attribute resolved to a Java class. This seemed more complex than warranted for carrying a single value. The added complexity here is in the form of a Java class that could be replaced by a simple attribute of the parent class. Making these values attributes of their parent element makes sense, because they would be resolved to instance variables of the parent class. Both the diagram in Figure 1 and the DTD in Appendix A reflect this change. For example, in the declaration section of a schema, if a variable is declared as a relation, then Drelation is used. The relational operator becomes the relOper attribute of the Drelation element. The same holds true for the Relation element in the formula section. Similarly, qualifier, unaryOper and operator are all attributes of the Formula, and unaryOper and operator are attributes of Expression. What Z calls

decorations on a schema name and on a variable are attributes of those elements, and the power-set indication is carried as the dpower attribute of the Declaration element. Having to change operator, relOper and from attributes of empty elements to attributes of the parent element was not a problem. The new arrangement is at least as good as what we had before.

5.3. Duplicate Elements in a Parent

Two appearances of an element in the same parent element also caused problems for JAXB. For example, to declare a variable as a relation of two types, the typeList element must appear twice in the Drelation element. Coding this causes JAXB to generate the Java class for Drelation with two declarations of the typeList instance variable. This, of course, is a Java syntax error. The same problem occurs in the Formula element when a relation is defined either between two variables or between two constants. JAXB should not generate syntactically illegal Java code when given XML that is well formed and valid for its DTD. The specification of two occurrences (or any specific number) is not the same as coding "*" to indicate zero or more, or coding "+" to indicate one or more occurrences. This problem can be circumvented in the DTD by defining a typeList2 element for use in the Drelation element, and by defining elements Variable2 and constant2 for use in the Relation element. In the long term, we would hope that JAXB provides a more elegant solution, since we were handing it a legal XML structure.

5.4. Combining Sequence and Choice

We encounter a far more difficult problem with elements that combine sequence and choice. The definition of the Relation element consists of a choice of two sequences. The definitions of the elements Declaration, Formula and Expression are each examples of a sequence consisting of one element and a choice of elements. The original definitions are included in the table of Figure 2. The problem with these complex elements is that JAXB makes no attempt to define Java classes for their component elements. Instead, JAXB simply generates getContent and setContent methods. For less complex elements, JAXB generates get and set methods for each of the component classes, except for those simple elements that resolve to instance variables. We would like JAXB to generate as much of the Java code required for building and traversing the internal trees as reasonably possible. The complexity of the Relation element can be sufficiently reduced by defining separate RelationV and RelationC elements for variables and constants, respectively. This increases the number of choices in the Formula element, but not its complexity. Figure 3 shows the changes resulting from adding the new constant, variable and relation elements.

```
<!ELEMENT Relation  ((Variable, Variable) | (constant, constant))>
```

```
<!ELEMENT Declaration (Variable+, (typeList | Drelation))>
```

```
<!ELEMENT Formula ( (Variable | Relation), Expression)>
```

```
<!ELEMENT Expression (Expression?, (Variable | Relation | Set |
   Sequence |   Bag | constant))>
```

Figure 2. Complex Elements Before Changes

The complexity of the other elements in Figure 2 is harder to reduce. Proper definitions in the JAXB binding schema can divide the sequence from the subordinate choice without introducing an intermediate element that would need to be coded in every XML document that represents a Z specification.

```
<!ELEMENT RelationV  (Variable, Variable2)>
<!ELEMENT RelationC  (constant, constant2)>
<!ELEMENT Declaration (Variable+, (typeList | Drelation))>
<!ELEMENT Formula ( (Variable | RelationV), Expression)>
<!ELEMENT Expression (Expression?, (Variable | RelationV | RelationC |
   Set | Sequence | Bag | constant))>
```

Figure 3. Complex Elements After Changes

5.5. Optional Element Bug

There is also a known and reported bug in JAXB that causes an optional element to be a required element. When an element listed in its parent is followed by a question mark, "?", then it is optional, but when code generated by JAXB is processing an XML document, the code fails with a "NullPointerException" if that optional element is not present. There are two instances of an optional element in the *ZML* DTD. The explain element in Spec is optional, and the recursive Expression element in itself is optional. A simple fix is to replace "?" with "*" in the parent element. This changes the definition from zero or one occurrence to zero or any number of occurrences. For the explain element, this is obviously not a problem. It certainly would hurt nothing to have comments broken into multiple parts. For the recursive nature of the Expression element, this is a minor problem at worst. Since the bug is known and we expect it to be fixed, we can allow this minor inaccuracy, temporarily. An alternative circumvention would be to modify the Java code generated by JAXB to test for a null pointer to the optional element before attempting to access it. More complex solutions are also possible by using eXtensible Stylesheet Language Transformations (XSLT) in combination with JAXB, but our goal is to find the least complex solution.

6. Explanation of the JAXB Binding Schema

JAXB takes most of the information it needs to generate the Java program code from the DTD, but the DTD does not contain all of the information needed. So, JAXB combines information from the DTD with information from the binding schema to build the Java classes that will be used to build and traverse the tree representing the XML document. JAXB requires a minimum binding schema, which contains only three tags. For this project, we add a few entries that provide additional information to JAXB about the more complex constructs that were left unresolved earlier. Specifically, the binding schema provides more information about the elements Declaration, Formula and Expression, from Figure 3, above, than JAXB can extract from the DTD.

The binding schema for *ZML* is shown in Figure 4. The first line contains the usual XML tag with the version number. There is a xml-java-binding-schema tag, which defines the type of XML document being defined. There is also at least one element tag to designate the root element in the XML document being processed. More than one root element can be designated.

In Figure 4, we have additional element tags, which help to define the more complex elements we described above. Consider the Declaration element. The DTD defines it as containing at least one Variable element, followed by a choice of either a typeList or a Drelation element. In Figure 4, the content tag within the Declaration element encloses a reference to the Variable element followed by a choice tag. The choice tag includes the property DecType, which becomes the name of the method that returns whichever element was chosen. The Java code that is traversing the needs only to invoke the "instanceof" method to determine whether a typeList or a Drelation element has been returned. The Formula element is similar, but with the choice tag coming first. When the contents of LeftSide are returned, they will either be the Variable element or the RelationV element. The Expression element is similar to the Declaration element, but with a longer list of possible elements when the RightSide contents are returned. These fairly simple entries in the binding schema allow the complex elements to be evaluated with a minimum of complexity in the Java code and no added XML tags in the Z specification.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xml-java-binding-schema version="1.0-ea">
 <element name="ZSpec" type="class" root="true"/>
 <element name="Declaration" type="class">
    <content>
       <element-ref name="Variable" />
       <choice property="DecType" />
    </content>
 </element>
  <element name="Formula" type="class">
    <content>
       <choice property="LeftSide" />
       <element-ref name="Expression" />
    </content>
 </element>
   <element name="Expression" type="class">
     <content>
        <element-ref name="Expression" />
        <choice property="RightSide" />
     </content>
  </element>
 </xml-java-binding-schema>
```

Figure 4. Binding Schema

7. Case Study: A Telephone Application

Included in this section are two examples of Z schemata encoded in XML, based upon the DTD described above. Both of these examples are taken from the Diller (1994) text. The first is a simple example from section 4.3.4 of the text. The schema PhoneDB is very simple. This example was chosen as the simplest example from the text, so the reader can see the structure of a Z schema encoded in XML without getting lost in details. Figure 5 is the schema as it appears in the text. It contains two declarations, one for a variable and the other for a relation. Its formula section contains only one simple formula.

PhoneDB

members: Π Person
telephones: Person φ Phone

---

dom telephones ζ members

Figure 5. (Diller, 1994)

Another example from the same book is recreated in Figure 6. This is only slightly more complex than the previous example, but it is complex enough to demonstrate the power of the *ZML* language we have created to express Z schemata. To this example, we have added a reference to the first schema, to show that it is needed.

AddEntry

ΔPhoneDB
members, membersə: Π Person
telephones, telephonesə: PersonφPhone
name?: Person
newnumber?: Phone

---

dom telephones ζ members
dom telephonesə ζ membersə
name? ε members
name?ϕnewnumber? ™ telephones
telephonesə = telephones Y {name?ϕnewnumber?}
membersə = members

Figure 6. (Diller, 1994)

The translation to XML is too large to include in-line, so it can be found in Appendix B. For completeness, Appendix B includes both the Figure 5 example and the Figure 6 example in a complete *ZML* specification document. The entire specification is enclosed between a ZSpec tag and its end tag. Within that, there are two Spec tags, one for each example. As explained above, the Spec tag was added to reduce the complexity of the ZSpec contents. Between Spec and its end

tag, we can have one explain tag for a comment, any required typeList tags, one for each list of variables being declared, and a single Schema tag.

Between the Explain and its end tag, any free form explanatory text can be written. The Z schema itself is sandwiched between the Schema tag and its end companion. The reader can examine Appendix B to see how the input decoration is carried on the “name?” and “newnumber?” variables as well as the prime decoration on “telephonesə” and “membersə”. After the schema name, which is not decorated, come two declarations. The first declaration is fairly simple, but we see the dpower attribute on the Declaration element, which designates power-set. The second variable is declared as a relation. We see there that Drelation is built from two variables with a relation operator as an attribute. The Formula portion of this schema is a single simple formula. The Formula element carries two attributes. The unaryOper is a unary operator for the left side, and the operator separates the two sides. The expression on the right consists only of a variable.

The second Spec tag includes only an explain element and a Schema element, showing that typeList is not required. The first two Declaration elements of this schema demonstrate that more than one variable can be declared at once. Two of the six Formula elements are worth discussing. The fourth Formula includes a relation on the left side. We see the RelationV element, because it is a relation of variables, with an attribute that indicates “related to”, with a Variable and a Variable2 element, both of which are decorated as input variables. The other notable example of the second schema is the penultimate Formula element, which exercises the recursive nature of the Expression element. That is, the expression on the right side of the formula consists of an expression, the union operator and a set. The inner expression consists only of a variable.

The point of using JAXB is to ease the writing of a Java application to process an XML document. A small example program called TestTrees is included in Appendix C. TestTrees performs three simple functions. First, it builds an internal tree from the example XML document, using the JAXB uumarshall method. Second, it performs an XML validation and prints out the same XML structure in a new file. The output XML file, ZSpec_new.xml, adheres to the rules specified in the DTD, because it passed the validation. The validation just demonstrates the correctness of the internal tree structure. The new file can easily be compared to the original to show that the two XML documents are structurally identical. Finally, the program locates and prints a few selected values from the internal tree. Figure 7 contains the printed output of the TestTrees program. The program writes the text from the "explain" elements, the schema name and any exSchema content to a file called ZSpec_out.txt. (The brackets around the “explain” text were added by JAXB to indicate text.) To properly display the Z symbols, use Netscape with the “Character Set” option “Unicode (UTF-8)” or Internet Explorer with “Encoding” option “Unicode (UTF-8)”. The method, called traverseTree is included to demonstrate how any and all values can be located and extracted from the internal tree that represents the XML document that has been read from the input file.

```
[Example from page 46 of the book]
PhoneDB
[Adding an Entry to the Database, page 48]
AddEntry
ΔPhoneDB
```

Figure 7. Output File

The results of this very small demonstration program are enough to convince us that we can also perform more complex processing on an XML Z specification, such as theorem proving or generating program code from a detailed design specification.

8. Summary of Recommendations to Improve JAXB

There are some limitations and problems with the JAXB tool that made it more difficult to use than expected. In this section, we recommend solutions to the JAXB limitations we encountered while using the tool, along with some specific recommendations for improving JAXB. In all cases, our governing philosophy is to keep the users' XML specification documents as simple as possible by making JAXB take on as much of the complexity as possible. We would like to keep the DTD as close to its original form as possible, but we are willing to take on a more complex DTD or a more complex binding schema in order to leave as little complexity as possible to our potential users.

8.1. Allow Nesting of Element

There is a need for JAXB to do a better job of addressing elements with nested constructs. In the project for this paper, we encountered three forms of nesting in elements. One has been described in the section "Nested Repetition", above. If we could have nested content tags in the binding schema, then we would not have needed to introduce the Spec tag. The second and third opportunities for nesting have been described in the section "Combining Sequence and Choice", above.

It would be very helpful if the binding schema could have elements nested within other elements. Perhaps an element tag could replace the element-ref tag within the content tag. Alternatively, the element-ref tag could be coded with a separate end tag and allow content tags to be coded between them. Some method is need for describing elements within elements.

8.2. Handle Multiple Occurrences of an Element

The problem with having multiple occurrences of an element in its parent element was described above, in the section "Duplicate Elements in a parent". In all of our examples, a parent element contains exactly two occurrences of the same kind of element. This would be a good use of arrays, if JAXB were made to support arrays. The result would be to avoid the need for typeList2, Variable2 and constant2 in the users' XML specification documents. If array support is not the answer, there should be some allowance for multiple appearances of some element within another element without generating illegal Java code.

8.3. Fix Optional Element Bug

The problem with an optional element, which is coded with "?" being required by JAXB is a known and reported defect. We expect this to be fixed soon. This issue does not require further explanation.

8.4. Insufficient User Documentation

The most obvious problem is the lack of good documentation. Two documents are offered for download along with the software. The smaller document is called the "User's Guide". If read sequentially, this document takes the reader through the steps required to generate and use the Java code for processing very simple XML documents. It uses a checkbook sample for which code is

included in the software download. The document is a good starting tutorial, but falls short of a complete user's guide.

The larger document is called "Working-Draft Specification". In this document, one might expect a full reference for the binding schema language, with syntax diagrams and explanations for all of the semantics. Unfortunately, this document is merely a larger user's guide. The section that is specifically dedicated to the syntax of a binding schema walks through a couple of examples that are only a little more complex than the example in the User's Guide. The syntax of each part and the parameters of the language can be found, but there are some parameters for which no explanation or only a partial explanation is provided. It would really help to have some more complex examples included in the User's Guide and to have a comprehensive reference manual.

Hopefully, more complete user documentation will appear as the tool progresses toward full product status. This should include a full language reference for the binding schema, and enough examples in the User's Guide to exercise most of the binding schema capabilities.

9. Conclusion

In this paper, we describe a DTD that defines the syntax of the Z specification language. We describe a Java program, written with the aid of JAXB, that demonstrates that our DTD usefully carries the Z specification so it can be passed among a variety of future tools. As a byproduct of using JAXB, we have described several limitations of the tool and our recommendations for improving it. Finally, we describe a case study that demonstrates the usefulness of our system, which we call ZML.

There were only two specialized tools used for this project. One is Sun's JAXB (Sun Microsystems, 2001), which was used heavily and was evaluated as part of this project. The intended purpose of JAXB seems to exactly match the needs of this project. In other words, it is a very good fit. Its specific limitations and problems are outlined in the previous section. The other tool used is IBM's Xeena (IBM, 2001), an XML editor and validity checker. It comes out of IBM's Alpha Works project. Xeena was used primarily to validate XML documents against the ZML DTD at different stages. It takes as inputs an XML document and the DTD upon which it is based. Xeena displays specific error messages and points to lines in the XML document that do not conform to the DTD. It also reports on where the XML document is not well formed, as well as errors in the DTD. The error messages have proved to informative and specific.

This project has demonstrated that a DTD can be created that defines the general form of a Z specification. It demonstrated that an XML document based upon that DTD could hold the contextual information of a Z specification. Finally, this project has demonstrated that a Z specification held in an XML document can be loaded by a Java program into an internal tree format and that tree traversed by the program to extract the information for any purpose.

Future work that builds on the work of this project could take any or all of the following directions. Create a Z editor that stores its results in XML form. Create a code generator that takes XML as its input. Modify a theorem prover to work with Z schemata in XML. Link any other tools that use Z as input and/or output.

Bibliography

Ankrum, T.S. & Appiah, S.O. (2001). "ZML: an XML Definition of Z Schemata". unpublished.

Diller, A. (1994). Z, An Introduction to Formal Methods. New York: John Wiley & Sons.

IBM (2001). "Xeena – XML Editing Environment, Naturally in Java!". Available: http://www.alphaworks.ibm.com/tech/xeena [2001, October 11].

Martin, A.P. (2002). "Community Z Tools Initiative (CZT)". Available: http://web.comlab.ox.ac.uk/oucl/work/andrew.martin/CZT/ [2002, December 26].

Spivey, J.M. (2001). "The Z Notation: a reference manual". Available: http://spivey.oriel.ox.ac.uk/~mike/zm/ [2001, February 27].

Sun, J, Dong, J, Liu, J, Wang, H. (2001) "Z Family on the Web with Their UML Photos". National University of Singapore. Available: http://nt-appn.comp.nus.edu.sg/fm/zml/ [2001, December 9].

Sun Microsystems, Inc. (2001). "The Java Architecture of XML Binding (JAXB)". Available: http://java.sun.com/xml/jaxb/index.html [2002, June 6].

W3C. (2002). "XML Core Working Group Public Page". Available: http://www.w3.org/XML/Core/ [2003, January 12].

Appendix A – DTD for Z Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--     T. Scott Ankrum      Scott_Ankrum@acm.org  -->
<!--     August 1, 2002                             -->
<!-- Z specs typically contain one or more typed variables, possibly -->
<!-- followed by an explanation and then a schema.  This grouping    -->
<!-- is repeated any number of times.  The Spec element exists to    -->
<!-- reduce complexity for JAXB.                                     -->

<!ELEMENT ZSpec (Spec*)>
<!ELEMENT Spec (typeList*, explain*, Schema)>
<!ELEMENT typeList (#PCDATA)>
<!ELEMENT typeList2 (#PCDATA)>
<!ELEMENT explain (#PCDATA)>
<!ELEMENT Schema (Name, ExSchema*, Declaration+, Formula+)>

<!ELEMENT Name (#PCDATA)>
<!ATTLIST Name dcorat (DELTA | XI) #IMPLIED>

<!ELEMENT ExSchema (#PCDATA)>
<!ATTLIST ExSchema dcorat (DELTA | XI) #IMPLIED>

<!ELEMENT Declaration (Variable+, (typeList | Drelation))>
<!ATTLIST Declaration dpower (PowerSet) #IMPLIED>

<!ELEMENT Variable (#PCDATA)>  <!-- variable being constrained or set -->
<!ATTLIST Variable dcorat1 (PRIME) #IMPLIED>
<!ATTLIST Variable dcorat2 (INPUT | OUTPUT) #IMPLIED>
<!ELEMENT Variable2 (#PCDATA)>  <!-- variable being constrained or set -->
<!ATTLIST Variable2 dcorat1 (PRIME) #IMPLIED>
<!ATTLIST Variable2 dcorat2 (INPUT | OUTPUT) #IMPLIED>

<!ELEMENT Drelation (typeList, typeList2)>  <!--relation/function type-->
<!ATTLIST Drelation relOper (lambda | mu | fcmp | dres | dsub | rsub | fovr |
cmp | inv | tcl | map | rel | tfun | tinj | pfun | tsur) #IMPLIED>

<!--  A Formula is a pre-condition, post condition or operation -->
<!ELEMENT Formula ( (Variable | RelationV), Expression)>
<!ATTLIST Formula qualifier (FOR-ALL | EXISTS | EXISTS-ONE | NOT-EXIST)
#IMPLIED>
<!ATTLIST Formula unaryOper (dom | ran | lnot) #IMPLIED>
<!ATTLIST Formula operator (equal | gt | lt| ge | le | ne | subset | subseteq |
mem | nmem) #REQUIRED>
<!-- A Formula may have a universal or existential qualifier, and maybe a unary
operator [or function]. -->
<!-- Then it will have a variable or relation, and operator and an expression or
formula -->

<!ELEMENT RelationV  (Variable, Variable2)>  <!-- relation or function variable
-->
<!ATTLIST RelationV relOper (lambda | mu | fcmp | dres | dsub | rsub | fovr |
cmp | inv | tcl | map | rel | tfun | tinj | pfun | tsur) #IMPLIED>
<!ELEMENT RelationC  (constant, constant2)>  <!-- relation or function variable
-->
```

```
<!ATTLIST RelationC relOper (lambda | mu | fcmp | dres | dsub | rsub | fovr |
cmp | inv | tcl | map | rel | tfun | tinj | pfun | tsur) #IMPLIED>

<!ELEMENT constant (#PCDATA)>
<!ELEMENT constant2 (#PCDATA)>

<!ELEMENT Expression (Expression*, (Variable | RelationV | RelationC | Set |
Sequence | Bag | constant))>
<!ATTLIST Expression unaryOper (dom | ran | lnot) #IMPLIED>
<!ATTLIST Expression operator (union | intersec | diff | subset | subseteq | mem
| nmem) #IMPLIED>
<!-- An expression is several expressions with operators between. -->

<!-- a set is defined by comprehension or enumeration -->
<!ELEMENT Set (Expression | constant* | RelationV* | RelationC*)>
<!-- a bag is defined by comprehension or enumeration -->
<!ELEMENT Bag (Expression | constant* | RelationV* | RelationC*)>
<!-- a sequence is defined by comprehension or enumeration -->
<!ELEMENT Sequence (Expression | constant* | RelationV* | RelationC*)>
```

Appendix B – XML for Example Schemata

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE ZSpec SYSTEM "ZSpec.dtd">

<ZSpec>
  <Spec>
    <typeList> Person, Phone </typeList>

    <explain> Example from page 46 of the book </explain>
    <Schema>
           <Name> PhoneDB </Name>
           <Declaration dpower="PowerSet">
             <Variable> members </Variable>
             <typeList> Person </typeList>
           </Declaration>
           <Declaration>
             <Variable> telephones </Variable>
             <Drelation relOper="rel">
                <typeList> Person </typeList>
                <typeList2> Phone </typeList2>
             </Drelation>
           </Declaration>

           <Formula unaryOper="dom" operator="subset">
               <Variable> telephones </Variable>
               <Expression>
                 <Variable> members </Variable>
               </Expression>
           </Formula>
    </Schema>
  </Spec>
  <Spec>
      <explain> Adding an Entry to the Database, page 48 </explain>
    <Schema>
         <Name> AddEntry </Name>
         <ExSchema dcorat="DELTA"> PhoneDB </ExSchema>
         <Declaration dpower="PowerSet">
           <Variable> members </Variable>
           <Variable dcorat1="PRIME"> members </Variable>
           <typeList> Person </typeList>
         </Declaration>
         <Declaration>
           <Variable> telephones </Variable>
           <Variable dcorat1="PRIME"> telephones </Variable>
              <Drelation relOper="rel">
              <typeList> Person </typeList>
              <typeList2> Phone </typeList2>
           </Drelation>
         </Declaration>
         <Declaration>
           <Variable dcorat2="INPUT"> name </Variable>
           <typeList> Person </typeList>
         </Declaration>
          <Declaration>
           <Variable dcorat2="INPUT"> newnumber </Variable>
```

```
          <typeList> Phone </typeList>
        </Declaration>

          <Formula unaryOper="dom" operator="subseteq">
              <Variable> telephones </Variable>
              <Expression>
                <Variable> members </Variable>
              </Expression>
          </Formula>

          <Formula unaryOper="dom" operator="subseteq">
              <Variable dcorat1="PRIME"> telephones </Variable>
              <Expression>
                <Variable dcorat1="PRIME"> members </Variable>
              </Expression>
          </Formula>

          <Formula operator="mem">
              <Variable dcorat2="INPUT"> name </Variable>
              <Expression>
                <Variable> members </Variable>
              </Expression>
          </Formula>

          <Formula operator="nmem">
              <RelationV relOper="rel">
                <Variable dcorat2="INPUT"> name </Variable>
                <Variable2 dcorat2="INPUT"> newnumber </Variable2>
              </RelationV>
              <Expression> <Variable> telephones </Variable> </Expression>
          </Formula>

          <Formula operator="equal">
              <Variable dcorat1="PRIME"> telephones </Variable>
              <Expression operator="union">
                <Expression>
                  <Variable> telephones </Variable>
                </Expression>
                <Set>
                  <RelationV relOper="rel">
                    <Variable dcorat2="INPUT"> name </Variable>
                      <Variable2 dcorat2="INPUT"> newnumber </Variable2>
                  </RelationV>
                </Set>
              </Expression>
          </Formula>

          <Formula operator="equal">
              <Variable dcorat1="PRIME"> members </Variable>
              <Expression>
                <Variable> members </Variable>
              </Expression>
          </Formula>

    </Schema>
  </Spec>
</ZSpec>
```

Appendix C –Example Java Program

```
/*  T. Scott Ankrum CSCI298 Summer 2002
 *     CSCI298 Summer 2002
 *  University of Maryland University College / George Washington University
 */
import java.io.*;
import java.util.*;
import javax.xml.bind.*;
import javax.xml.marshal.*;

public class TestTrees {

    public static ZSpec internalTree = new ZSpec();
    public static final char delta = '\u0394';
    public static final char xi = '\u039E';

    public static void main(String[] args) throws Exception {
   buildTrees();  // Build the content trees
   validateAndMarshalTrees();
   traverseTree();
   } // main

   public static void traverseTree() throws Exception {
         Spec specGroup = new Spec();
         Schema schema = new Schema();
         Name name = new Name();
         ExSchema external = new ExSchema();
      // Create an output file for the formatted spec.
        FileOutputStream specFile = new FileOutputStream("ZSpec_out.txt");
        OutputStreamWriter specOut = new OutputStreamWriter(specFile, "UTF8");

       List specification = internalTree.getSpec();
       for (ListIterator i = specification.listIterator(); i.hasNext();) {
         specGroup = (Spec) i.next();
         schema = specGroup.getSchema();
         if (schema == null) break;
         specOut.write(specGroup.getExplain().toString() +"\n");
         name = schema.getName();
         specOut.write(name.getContent().toString() +"\n");
         List exRefs = schema.getExSchema();
         for (ListIterator j = exRefs.listIterator(); j.hasNext();) {
           external = (ExSchema) j.next();
           char greek = ' ';
           if (external.getDcorat().toString().equals("DELTA")) greek = delta;
           if (external.getDcorat().toString().equals("XI"))   greek = xi;
           specOut.write(greek + external.getContent().toString() +"\n");
         } // for j
       } // for i
     specOut.close();
   } // traverseTree

   public static void buildTrees() throws Exception {
         // Unmarshal the ZSpec.xml file
      File zSpecIn = new File("ZSpec.xml");
      FileInputStream fNewIn = new FileInputStream(zSpecIn);
```

```
         // Unmarshal the XML file to a tree
      try {
          internalTree = internalTree.unmarshal(fNewIn);
         }
      finally {
             fNewIn.close();
            }
    } // buildTrees

   // Validate the updated tree and marshal it
   public static void validateAndMarshalTrees() throws Exception {
      internalTree.validate();
         // Create an output file for the updated formal spec
      File zspec_new = new File("Z-Spec_new.xml");
      FileOutputStream fCOut = new FileOutputStream(zspec_new);
         // Marshal the updated formal spec
      try {
             internalTree.marshal(fCOut);
         }
      finally {
             fCOut.close();
            }
   } // validateAndMarshalTrees

} // TestTrees
```