

# ZML: XML Support for Standard Z

Mark Utting<sup>1</sup>, Ian Toyn<sup>2</sup>, Jing Sun<sup>4</sup>, Andrew Martin<sup>3</sup>, Jin Song Dong<sup>4</sup>,  
Nicholas Daley<sup>1</sup>, and David Currie<sup>5</sup>

<sup>1</sup> The University of Waikato, Hamilton, NZ  
{marku,ntd1}@cs.waikato.ac.nz

<sup>2</sup> The University of York  
ian@cs.york.ac.uk

<sup>3</sup> Oxford University  
Andrew.Martin@comlab.ox.ac.uk

<sup>4</sup> The National University of Singapore  
{sunjing,dongjs}@comp.nus.edu.sg

<sup>5</sup> IBM UK Labs, Hursley Park, Winchester, Hants, UK  
david\_currie@uk.ibm.com

**Abstract.** This paper proposes an XML format for standard Z. We describe several earlier XML proposals for Z, the problems and issues that arose, and the rationales behind our new proposal. The new proposal is based upon a comparison of various existing Z annotated syntaxes, to ensure that the mark-up will be widely usable. This XML format is expected to become a central feature of the CZT (Community Z Tools) initiative.

## 1 Why an XML format for Z?

The publication during 2002 of the ISO Z Standard [3] represents a significant milestone for the development and interoperability of Z tools. It has established what notation should be exchanged, but not necessarily how. Technology has advanced during the development of the standard, so it now seems most natural for tools to interact using an XML mark-up [9].

This paper describes such a mark-up, intended to be a development of the Standard's work, as a contribution to the Community Z Tools<sup>1</sup> (CZT) initiative. CZT has been proposed in response to the observation that many interesting Z tools have been developed, but few have built large user communities, and many have found it necessary to invest disproportionately large amounts of effort in the relatively mundane activities of parser and pretty-printer development. The initiative aims to define interfaces and interchange facilities (and later, code libraries) which Z tool developers can draw on in an open-source spirit, with the aim both of promoting interoperability and of relieving those wishing to develop novel tools for visualisation, animation, refinement, proof, and so on, from the need to invest effort in the user interface code.

XML is a development, like HTML, from the SGML [4]. Early drafts of the Z Standard included an SGML mark-up, but it was found hard to maintain. XML now

---

<sup>1</sup> See <http://web.comlab.ox.ac.uk/oucl/work/andrew.martin/CZT>.

enjoys a much wider take-up than SGML, having quickly become a new standard for structured information interchange between tools.

Without such a mark-up, Standard Z allows specifications to be exchanged using Unicode (UCS[1,2]). However, this representation is suitable only for interchanging raw (unparsed) Z specifications, without annotation. Tools (and sometimes authors) benefit from being able to annotate terms with type information, anticipated usage and refinement targets, free-form comments, and so on. A particular presentation (on paper, on screen, or within program data structures) may make use of some of these annotations and discard others. An XML format facilitates the inclusion of such annotations, with as little or as much structure as is appropriate. In the longer term, when this use of XML reaches greater maturity, we would expect the format described here to become part of the ISO Z Standard.

### 1.1 Requirements of a Z Interchange Mark-Up

We have three requirements for an XML mark-up for Z.

*Annotations.* The already mentioned annotations should be accommodated in the interchange mark-up wherever tools wish to put them. The forms of individual annotations should not be constrained. There should be some pre-defined annotations for types and for source-file locations (so that error messages can refer to the source of an error), but it should also be possible for tools to define additional annotations. Tools that do not understand such annotations should simply ignore them.

*Injectivity.* The concrete syntax of Z provides different ways of writing the same things. For example, a boxed schema paragraph may be written in an equivalent definitional form, without the box. After a specification has been transferred between tools, the user wants to be reassured as much as possible (by avoiding unexpected changes of presentation) that their specification document has not been changed. Consequently, the interchange mark-up for Z should capture sufficient information from the concrete representation to be able to resurrect the same concrete phrases (though not necessarily the same layout). In other words, we want the conversion of a textual Z specification into XML format to be one-to-one (injective), so that the concrete representation before and after interchange, ignoring annotations, remains recognisably the same. For the schema paragraph example, this means keeping a note of whether or not the boxed representation is used. In this paper, we avoid using the traditional term *abstract syntax* because of this avoidance of loss of information from the concrete form.

*Commonality.* Conversely, for reasons of simplicity and demonstrable soundness, tools should need to deal with as few cases as possible. This implies that we should merge equivalent concrete constructs whenever possible (the Z standard has a large number of transformation rules that do exactly this). For example, a tool might offer to display the signature of a schema paragraph regardless of whether or not it is boxed. This is easier if a common *annotated syntax* is used for both of the concrete representations of the schema paragraph. Interchange will be eased if the mark-up is based on an annotated syntax that identifies similar commonalities to those exploited by tools. In this paper, we use two approaches to merging constructs while preserving injectivity:

1. using a common XML tag for two similar constructs, but adding attributes to distinguish between the constructs;
2. using distinct XML tags and adding a common type hierarchy above them to reflect their commonality.

The second approach has an additional advantage: the type hierarchy of commonality is similar to a typical inheritance hierarchy in object-oriented programs, which makes it easier to map between the XML structure and Java or C++ classes. This is useful, because one of the CZT aims is to develop a Java library for building Z tools.

The annotated syntaxes used within existing tools have already addressed these issues of annotations, injectivity and commonalities. The annotated syntax used within Standard Z addresses some of these issues. An interchange mark-up for Z will be easier for a tool to use if the mark-up is similar to the tool's own annotated syntax, but there is considerable variation between existing tools.

This paper compares some existing annotated syntaxes and describes an XML mark-up based on their common features or best features. We aim to define a mark-up that will be usable not only by proposed CZT developments but also by developments of existing tools. Hence, we are interested to receive feedback from other tool builders.

## 1.2 Specifying the XML Structure: DTD or XML Schema?

There are many different ways in which Z specifications could be expressed in XML. To specify exactly which structures of XML we propose to use, and the well-formedness conditions on those structures, we need to specify a particular subset of XML. Such a specification is typically written in either of two languages: as a *Document Type Definition* (DTD), or as an *XML Schema*. The provision of such a specification allows a *validating parser* to perform more accurate well-formedness checking, and is useful to toolbuilders for defining what tools should be able to interchange.

The DTD notation is older and simpler than the XML Schema notation, and more human-readable, but the XML Schema language allows better specification of the data types of elements than the DTD language. XML Schema has many built-in datatypes such as string, integer, boolean, float, date, time and so on, and provides mechanisms to constrain the allowable content of an element or attribute, such as setting a valid range of values or defining a regular expression to which the content must conform. New types can be defined from scratch or by constraining or extending an existing type. This allows hierarchies of complex types to be constructed. Furthermore, XML Schemas are themselves written in XML. This makes the document descriptions more verbose, but also far more extensible than they were in the original DTD syntax. Declarations can have richer and more complex internal structures than declarations in DTDs. Thus XML Schemas can be stored along with other XML documents in XML-oriented data stores, referenced, and even styled, using tools like XLink, XPointer, and XSLT<sup>2</sup>. For our purposes, we prefer to use XML schema notation, to obtain a tighter specification of the structure, and to take advantage of XML tools, such as XSLT.

<sup>2</sup> See [www.w3.org](http://www.w3.org).

## 2 Previous Work

There were earlier attempts to define XML mark-ups for Z [16,13] but these did not support the interchange of annotations such as the types of expressions, and were based on an earlier version of Z, described by Spivey [12]. For example, Z/EVES [11] supports an XML mark-up for communication between tools, based on Spivey Z.

Before ZB2002, Toyn wrote a DTD for Standard Z, influenced by the abstract syntaxes of CADiZ and Zeta. This DTD has heavily influenced our proposal in this paper.

For example, here is the top-level element declaration from that DTD:

```
<!ELEMENT Z:Spec; (((Z:Sect;*), Z:SpecAnns;?) | Z:PCDATA)>
```

This defines a Z specification to be either a sequence of sections followed by an optional *specification annotations* element, or a PCDATA alternative, which is another element that is defined to contain just #PCDATA (*parsed character data*). Every element in the DTD includes a Z:PCDATA alternative, so that if one part of a specification contains an error, the whole specification can still be passed between tools. For example, a fully-parsed specification might be passed to an editor, and after editing is complete, the editor might pass it back with unchanged portions still in parsed form, but the edited portions in Z:PCDATA form.

During 2002, David Currie validated the DTD, and Utting and Daley manually derived a Java class hierarchy from it [5]. During this process, we identified several difficulties with the DTD structure:

1. The presence of an ‘unparsed’ alternative for *every* element allowed extremely fine-grained portions of the specification to be left unparsed, but dramatically complicated the Java class hierarchy. Basically, every element E of the DTD had to be translated into *three* Java classes: an abstract class E and two concrete subclasses, EParsed and EUnparsed, where EParsed contained fields that matched the parsed structure and EUnparsed contained just an unparsed string. The real disadvantage of this was that every piece of Java code that accessed an E object had to immediately check whether it was parsed or unparsed. This issue was not specific to Java, but would affect processing in every language.

To solve this problem, our new proposal in this paper limits the *granularity* of the unparsed portions so that an entire paragraph (for example, one schema) is the smallest unparsed portion allowable. This simplifies processing, because it means that only the top-level of processing needs to consider unparsed portions and once we see a parsed paragraph, we know that everything inside it will also be parsed.

2. Each element in the DTD has its own kind of annotation, as illustrated in the example above (SpecAnns). Each kind of annotation is given a default definition in the DTD (expression annotations contain just a type, schema annotations contain a signature etc.), but can be overridden by providing an extended DTD that adds extra fields. However, because each kind of expression has its own kind of annotation, it is necessary to override all 23 kinds of expressions to add a new annotation to expressions (or 7 kinds for predicates, 5 kinds for paragraphs, etc.).

To solve this problem, and make it easier to add new kinds of annotations, we have changed to a more loosely-typed view of annotations that is similar to the annotations

in Zeta. Each Z construct can contain a list of arbitrary annotations. This means that a tool could attach an annotation to an inappropriate construct (such as putting a type annotation on a predicate), but such annotations do no harm and can simply be ignored. On the other hand, there are many kinds of annotations (such as hyperlinks, source-code positions and comments), that we want to be able to attach to arbitrary constructs, and this is easier with these loosely-typed annotations.

3. In an object-oriented class hierarchy, it is possible to organise the hierarchy to reflect commonality, so that common fields and methods can be inherited. This is more flexible than the DTD structure, which does not have any kind of inheritance. This resulted in more differences between the DTD structure and our ideal Java class hierarchy than we would have liked.

Our new proposal solves this problem by using XML Schema to specify the structure of the Z mark-up. XML Schema offers a rich set of (single) inheritance features between types, as well as a *substitution group* facility which is similar to subtyping in object-oriented languages.

Also in 2002, at the National University of Singapore, Dong and Sun developed a new version of their XML Schema, based more closely on the annotated syntax structure of the Z standard. This did not support unparsed alternatives or annotations, and made less use of commonalities than Toyn's DTD, but it included extensions for supporting Object-Z and TCOZ (Timed Communicating Object Z) [10]. They demonstrated that it is possible to use the XSLT transformation language to transform the XML form of Z into elegant HTML with proper boxes and mathematical symbols.<sup>3</sup> The generated HTML includes cross-references, and buttons for expanding schema expressions and folding them again. The impressive feature is that this transformation system actually runs in your own browser, using standard technologies (XML, XSLT and Unicode).

This shows the promise of our XML proposal—it allows one to download a parsed and type-checked Z specification in an XML format that is ideal for importing into tools, yet still view it and explore it (following cross references etc.) with a standard web browser.

Furthermore, Dong and Sun have defined an XSLT stylesheet for automatically transforming the Object-Z/TCOZ models in XML into UML class diagrams [14]. The XSLT encodes the projection rules from the formal notations into their corresponding UML counterparts. Recently this work has been extended to support the auto-generation of UML statechart diagrams from Object-Z/TCOZ specifications via Java XML parser [6]. Both implementations take the customized XML format as a standard input and performs XML transformation into XMI (XML Metadata Interchange) format for visualization. In addition, an XML-based type checker was built for the static type checking of Z/Object-Z/TCOZ specifications in XML format.

### 3 Influences on Our Design

The structure of our proposed XML mark-up is based on Toyn's DTD, which was designed by comparing and merging the best features of three annotated syntaxes used

<sup>3</sup> See <http://nt-appn.comp.nus.edu.sg/fm/zml> for a demonstration of this system. It requires an appropriate Unicode font on your computer, such as Microsoft Arial Unicode.

by the Z standard, CADiZ and Zeta. This section briefly describes each of these systems and how they differ from our goals.

### 3.1 Standard Z

Standard Z's annotated syntax provides the basis for its definition of the type system and semantics of Z. These are the only functions defined on its annotated syntax. In particular, the standard has no need to resurrect concrete syntax. It has annotations for types of expressions, signatures of paragraphs, and section-type environments of sections. Commonalities are identified by *syntactic transformation rules*, which define the translation of concrete syntax to equivalent annotated syntax. Some of these rules are quoted below.

XML mark-up differs from Standard Z's annotated syntax because of the need to resurrect concrete syntax and the need to support a greater variety of functions and annotations.

### 3.2 CADiZ

CADiZ's annotated syntax supports typechecking, prettyprinting (i.e. resurrection of concrete syntax), interactive browsing (i.e. tracking of references to declarations and inspection of types, signatures and environments), and logical inference (i.e. transformation to equivalent notation, as in the course of proofs). Z notation is also used as patterns in tactics for automated reasoning.

In CADiZ's annotated syntax, the representation of declarations plays many roles. As well as representing the name and expression of a declaration, it records the declared variable's type, allowing signatures to be represented as lists of declarations, and it records which expressions refer to it. An inclusion declaration brings new copies of a declaration into scope, so that uses of the included declaration are not confused with uses of the original declaration. Expressions record the declarations to which they refer—this supports interactive browsing. They also support logical inference rules, correctly handling variable capture side-conditions: the inference rules maintain bindings of references to declarations, and the prettyprinter does renaming wherever variable capture would otherwise seem to occur. The representation of declarations causes CADiZ's annotated syntax to be not a tree structure but a more general graph, which would be inconvenient for a textual interchange mark-up such as XML (but more on this later).

CADiZ[15] can be said to support Standard Z—the deviations are very minor. (It does have some extensions to Standard Z, but we will ignore those.) CADiZ's annotated syntax is not fixed, and has changed frequently in the past (and may change in the future to be closer to this proposal).

### 3.3 Zeta

Zeta's annotated syntax supports typechecking, prettyprinting (i.e. resurrection of concrete syntax), and animation (i.e. automatic reduction of expressions). Those are the functions of the core edition of Zeta.

XML mark-up differs from Zeta's annotated syntax wherever Zeta[8] deviates from Standard Z.

### 3.4 Standard Terminology

The main syntactic rules (Specification, Section, Paragraph, Predicate and Expression) are present in all annotated syntaxes for Z, though not with the same names. In some tools, this renaming reflects the widening of syntactic rules to include non-Z phrases. The following table summarises these names, and suggests names to be used for the elements in XML. The Z: prefix is just a namespace prefix, and can be omitted in XML documents whose default namespace is our XML Schema. We use a postfix \* symbol to indicate possible repetition of a construct (zero or more times) and + to indicate one or more repetitions.

Standard Z	CADiZ	Zeta	XML
Specification	doc*	UnitAbsy*	Z:Spec
Section	doc	UnitAbsy.Section	Z:Sect
Paragraph	def	Item	Z:Para
Predicate	pred	Predicate	Z:Pred
Expression	term	Expr	Z:Expr

## 4 Our XML Schema Proposal

In this section, we go through each major construct of the Z notation, briefly comparing the Z standard, CADiZ and Zeta, and describing our proposed XML structure. The XML Schema was developed and validated using the XML-Spy tool<sup>4</sup>, and the diagrams were also partly generated with XML-Spy. The diagrams use two connectors: the three-dots connector defines a *sequence* of the elements on its right, while the three-way switch connector defines a *choice* between the elements on its right. Dashed lines indicate optional components—this is usually obvious from the repetition counts, like  $0 \dots \infty$ , below the optional constructs.

### 4.1 Specifications and Sections

Standard Z specifications are either anonymous or sectioned. The standard syntactically transforms anonymous specifications to sectioned specifications, as follows (Z standard, clause 12.2.1.1).

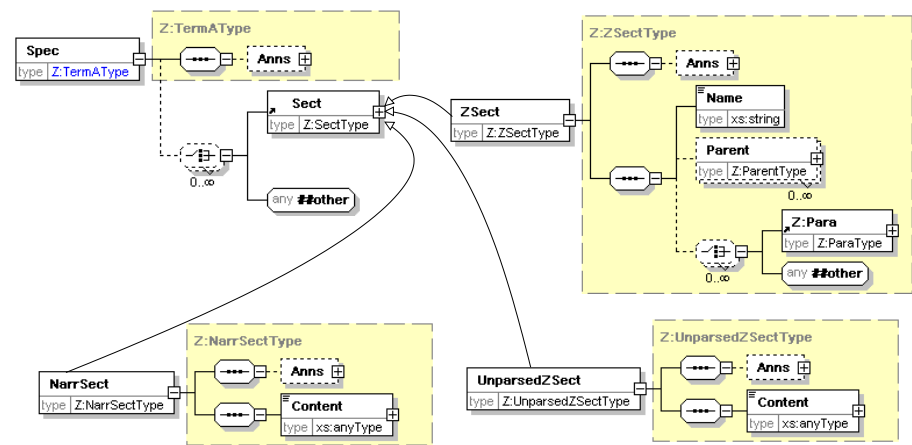
$$D_1 \dots D_n \implies \textit{Math toolkit ZED section Specification parents standard\_toolkit END } D_1 \dots D_n$$

The name *Specification* can be anything distinct (CADiZ uses the name of the file that the specification came from). To allow the concrete syntax to be resurrected precisely, it is necessary to know whether a section was originally anonymous—we do this by associating a Boolean attribute *Anon* with each section.

So, a specification can be represented as just a sequence of sections, and both CADiZ and Zeta use that representation. The following table lists the components of a Z section.

<sup>4</sup> See [www.xmlspy.com](http://www.xmlspy.com)

Standard Z	CADiZ	Zeta	XML
Section	doc	UnitAbsy.Section	Z:Sect
NAME	word	Name	Z:Word
seq NAME	parent*	Name*	Z:Word*
seq Paragraph	def*	Item*	Z:Para*
SectTypeEnv			Z:Anns/Z:SectTypeEnvAnn



**Fig. 1.** XML structure for an entire Specification. The arrows pointing towards **Sect** indicate that **ZSect**, **UnparsedZSect** and **NarrSect** are in the **Sect** substitution group, so each **Sect** element can be replaced by any one of them.

Fig. 1 shows a diagrammatic presentation of the corresponding XML structure, omitting some details such as attributes. It shows that a specification is a sequence of zero or more constructs, where each construct is either a parsed section (**ZSect**), an unparsed section (**UnparsedZSect**), a narrative portion (**NarrSect**) or some other kind of arbitrary (non Z-related) XML element (the **any ##other**). Each parsed **ZSect** section must be a sequence of an optional set of annotations, then a name, then zero or more parents, then zero or more paragraphs (or other XML elements). The top-level **Spec** element also has three optional attributes (not shown) to record its *Creator* and the *Date* and *Time* of the last modification. Note that inside an **Anns** tag, any XML elements are allowed—our XML proposal pre-defines several annotations, but other tools are free to define more. We have set *processing* = *lax* within the **Anns** element, which means that Z tools and other validation tools should simply ignore any annotations they do not understand.

Within a **ZSect**, the list of parent names need not include *prelude*, as that is implicitly a parent of all sections. If there are no parents, the **ZSect** element does not record whether or not the keyword *parents* occurred in the concrete representation. This doesn't matter sufficiently to deserve the declaration of an attribute.



*Support for Z Extensions.* There have been numerous extensions of Z in the past, and this will probably continue. Furthermore, within a Z specification, we want to allow complementary kinds of specification, such as CSP specifications, UML diagrams, or new kinds of paragraphs defined by some extension of Z like Object-Z or TCOZ. Fig. 1 shows that, within specifications and sections, our XML mark-up allows arbitrary elements from *other* namespaces to be interspersed with Z constructs. This means that the XML tags that belong to the standard Z namespace will be checked and processed by Z tools, while text and unknown tags (from other namespaces) will be ignored. In other words, the formal Z constructs (sections and paragraphs) are viewed as being part of a larger narrative, which may contain other kinds of top-level mark-up. This is a more permissive, egalitarian style of mark-up than allowing only standard Z constructs to appear at the top level.

## 4.2 Paragraphs

Toyn's DTD defined `Z:Para` to be a *choice* between six kinds of paragraph. XML Schema gives us several different ways of doing this, and we have decided to use a newish XML Schema feature, *Substitution Groups*, rather than choice groups, because substitution groups are similar to an object-oriented subtyping structure (where a subtype object can replace a supertype object), and can support inheritance of attributes and elements.

Substitution groups make it easy to extend the structure. For example, a Z extension can add a new kind of paragraph simply by defining a new element with `substitutionGroup="Para"`. It is also easy to add new features to one of the subtypes, like `AxPara`, by declaring a new element whose type extends or restricts the type of `AxPara` and says `substitutionGroup="AxPara"` (the substitution relationship is transitive).

Here is the XML Schema definition for `Para`. It is declared to be abstract so that XML files *must* contain a more specific kind of paragraph, wherever a `Para` element is expected.

```
<xs:element name="Para" type="ParaType" abstract="true"/>
```

The following subsections go through each kind of paragraph, describing their structure.

**Given Types Paragraph.** The following table lists the components of a given types paragraph.

Standard Z	CADiZ	Zeta	XML
Given types Paragraph	givdef	Item.AxiomaticDef*	Z:GivenPara
seq NAME	dec*	Expr.GivenType	Z:DeclName*
Signature			Z:Anns/Z:TypeEnvAnn

In CADiZ, all declarations (given types, generic parameters, variables) share the same dec representation. This has the advantage of providing a basis for tracking all references to each declaration.

In Zeta, a given types paragraph is represented as an `Item.AxiomaticDefs` sequence, in which each `Item.AxiomaticDef`'s expression is an `Expr.GivenType` containing the name of a given type. This is an instance of a more general approach: Zeta represents each Z global definition as an `Item.AxiomaticDef`, using additional kinds of expressions beyond those of Standard Z to make this possible. Concretely, a given types paragraph (or a single given type) is not an expression, and so Zeta's representation seems a bit forced.

In XML, a given types paragraph is marked-up using the `Z:GivenPara` element, whose type is shown in Fig. 2. To save space, we do not show the annotation elements (`Anns`) in this and future diagrams, because they appear on virtually all constructs.

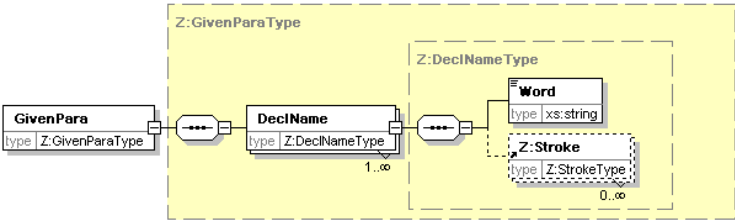


Fig. 2. XML structure for Given Type paragraphs

**Axiomatic Description Paragraph.** The following table lists the components of an axiomatic description paragraph.

Standard Z (Generic) axdef Paragraph	CADiZ axidef	Zeta Item.AxiomaticDef	XML Z:AxPara
seq NAME	dec*	NameDecl*	Z:DeclName*
Expression	sch	Expr.Text	Z:SchText
Signature			Z:Anns/Z:TypeEnvAnn

In CADiZ and Zeta, non-generic axiomatic description paragraphs are represented as generic ones with an empty list of generic parameters. Standard Z differs, as it was thought that the semantics of generics would be easier to understand if the semantics of non-generics were defined separately first.

The declarations and predicate parts of an axiomatic description paragraph are represented differently in the different annotated syntaxes. Standard Z transforms them to an expression. CADiZ retains the schema text, represented by a distinct rule in the annotated syntax. Zeta views the schema text as an expression. We believe that some annotations can usefully be placed on schema texts, and that any single expression appearing where a schema text is expected is best represented as an inclusion in a schema text, so that there is somewhere to record those annotations.

Fig. 3 shows our XML structure for the `AxPara` element, as well as for schema text and declarations. Note the three ‘subtypes’ of `Decl1`. These are all declared as belonging to the `Decl1` substitution group so that they can appear wherever a `Decl1` is required.

The following definitions from the Z standard (syntactic transformations 12.2.3.1—12.2.3.4) show how to represent (generic) schema definition paragraphs and (generic) horizontal definition paragraphs as (generic) axiomatic description paragraphs. The *SCH*, *END* etc. are box tokens, which abstract away from the exact appearances of paragraph outlines.

$$\text{SCH } i \ t \ \text{END} \implies \text{AX } [i == t] \ \text{END}$$

$$\text{GENSCH } i \ [i_1, \dots, i_n] \ t \ \text{END} \implies \text{GENAX } [i_1, \dots, i_n] \ [i == t] \ \text{END}$$

$$\text{ZED } i == e \ \text{END} \implies \text{AX } [i == e] \ \text{END}$$

$$\text{ZED } i \ [i_1, \dots, i_n] == e \ \text{END} \implies \text{GENAX } [i_1, \dots, i_n] \ [i == e] \ \text{END}$$

Generic operator definition paragraphs have their operator names syntactically transformed to ordinary names (syntactic transformations 12.2.9.1—12.2.9.4) and hence they become generic horizontal definition paragraphs that can be represented as generic axiomatic description paragraphs.

To support resurrection of the original concrete representation, we add an attribute *Box* with values: *OmitBox*, *AxBBox* (the default), or *SchBox*. A further Boolean attribute called *Mixfix*, distinguishes whether mixfix syntax is used in the definition of a generic operator e.g.  $\_ \leftrightarrow \_ [X, Y] == \mathbb{P}(X \times Y)$ .

**Free Types Paragraph.** The following tables list the components of a free types paragraph.

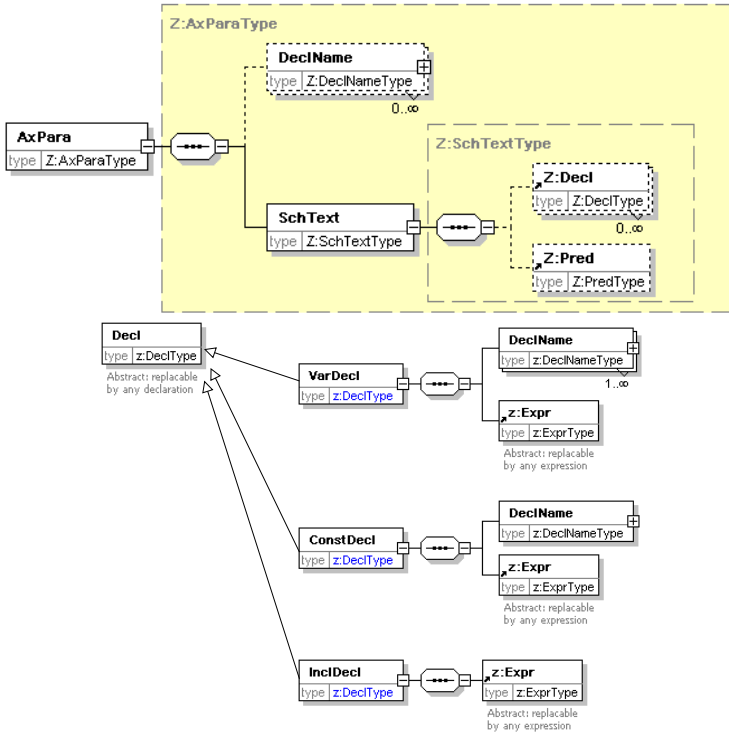
Standard Z	CADiZ	Zeta	XML
Free types Paragraph	datdef	Item.AxiomaticDef*	Z:FreePara
seq Freetype	fret+	Expr.FreeType	Z:FreeType+
Signature			Z:Anns/Z:TypeEnvAnn

In Zeta, the representation of free types paragraphs is similar to that of other global definitions (see the earlier discussion in the Given Types section).

Standard Z	CADiZ	Zeta	XML
Freetype	fret	Expr.FreeType	Z:FreeType
NAME	dec	NameDecl	Z:DeclName
seq Branch	bra+	Branch+	Z:Branch+

The representation of a branch is very different in different tools, and so cannot readily be tabulated.

Standard Z	XML
Branch	Z:Branch
NAME	Z:DeclName
Expression	Z:Expr?



**Fig. 3.** XML structure for Axiomatic Definition paragraphs, Schema Text and Declarations. The arrows pointing towards `Decl` indicate that `VarDecl`, `ConstDecl` and `InclDecl` are in the `Decl` substitution group, so each `Decl` element can be replaced by any one of them.

In  $\text{CADiZ}$ , a Branch's name and optional expression are both represented by a single `dec` value, allowing references to the name to be tracked.

In Zeta, a Branch is either a Constant or a Function. A Constant has just a `NameDecl`, whereas a Function has both a `NameDecl` and an `Expr`.

In XML, a free types paragraph is marked-up using the `Z:FreePara` element, whose type is shown in Fig. 4.

**Conjecture Paragraph.** Standard Z conjectures have a single consequent predicate and zero or more generic parameters.

Zeta does not support conjecture paragraphs.

In  $\text{CADiZ}$ , conjectures are represented as particular cases of a more general syntax for sequents. Sequents allow for zero-or-more generic parameters, zero-or-more levels of nested `DeclParts`, zero-or-more antecedent predicates, zero-or-more consequent predicates, and a name for the sequent. This more general syntax assists humans doing proofs interactively, but adds nothing semantically: any sequent can be rearranged into an equivalent single-consequent form that conforms to the Z standard (ignoring the se-

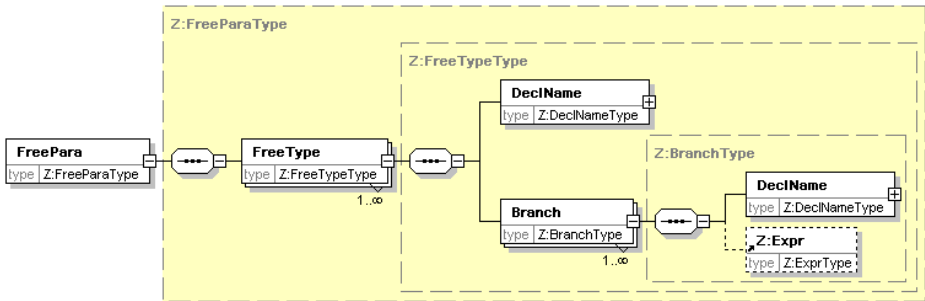


Fig. 4. XML structure for Free Type paragraphs

quent’s name, which can be thought of as an annotation). Other reasoning tools for Z may use different representations for sequents. So it seems inappropriate to define an XML mark-up for anything more complicated than a Standard Z (generic) conjecture.

The following table lists the components of a conjecture paragraph.

Standard Z	XML
(Generic) conjecture Paragraph	Z:ConjPara
seq NAME	Z:DeclName*
Predicate	Z:Pred
Signature	Z:Anns/Z:TypeEnvAnn

In XML, a conjecture paragraph is marked-up using the `Z:ConjPara` element (Fig. 5). This representation suffices for both generic and non-generic conjecture paragraphs: the sequence of generic parameters is empty in the non-generic case.

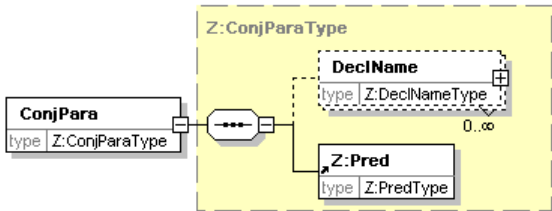


Fig. 5. XML structure for Conjecture paragraphs

**Operator Template Paragraph.** Standard Z has operator template paragraphs in its concrete syntax but not in its annotated syntax, because they affect how the specification is parsed but have no further meaning themselves. To be able to interchange them and resurrect their concrete syntax, and the concrete syntax of the operators they define, the XML mark-up must provide a representation of them.

Operator templates are one of the innovations of Standard Z and were subject to some late changes, so tools are unlikely to support operator templates exactly as in Standard Z (excepting CADiZ). The concrete syntax allows explicit declaration of precedence and associativity only for infix function and infix generic operators. Other operators have implicit precedences and associativities, which it is convenient to make explicit in the annotated syntax.

The following table lists the components of an operator template paragraph.

Standard Z	CADiZ	Zeta	XML
Operator template Paragraph	fixdef	Fixity	Z:OptempPara
Category	cat	isGeneric	Z:Cat (Attr)
Prec	nat	prio	Z:Prec (Attr)
Assoc	boole	?	Z:Assoc (Attr)
Template	(nat,word)+	Component*	See Fig. 6

In CADiZ, a Template is represented as a list of pairs. While this enforces alternation of operators and operands, it may unfortunately appear to add an unwanted operand at the beginning and/or an unwanted operator at the end, for which distinguishable values are needed to avoid confusion.

In Zeta, a Template is represented as a list of Components. Each Component is either a Keyword, Operand or OperandList. Zeta appears to parse declarations of associativity, but it does not appear to keep a representation of associativity in its annotated syntax. Its annotated syntax also appears not to distinguish relation and function categories.

In XML, an operator template paragraph is marked-up using the Z:OptempPara element (Fig. 6). In addition, each Z:OptempPara has three attributes:

- Cat (category) which can equal Relation, Function or Generic.
- Assoc which can be Left or Right.
- Prec (precedence) which is a natural number.

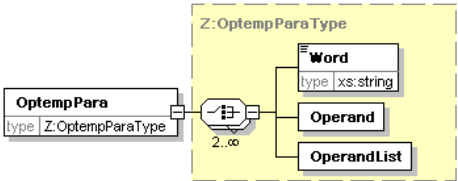


Fig. 6. XML structure for Operator Template paragraphs

**Narrative Paragraph.** To allow natural language narrative to appear between Z paragraphs, we define a NarrPara element, containing annotations and a Contents element which contains arbitrary unicode and markup. This is similar to NarrSect in Fig. 1.

**Unparsed Paragraph.** Our final kind of paragraph does not appear in Zeta or the Z standard, because their annotated syntax representations are used only *after* an entire specification has been successfully parsed. However, since our XML format may be our source representation, we need to be able to represent erroneous (unparsable) specifications as well. Similar to the `ErrorDef` paragraph in `CADiZ`, we use a special paragraph called `UnparsedPara`, whose structure is the same as `UnparsedZSect` (see Fig.1). If a tool attempts to parse an `UnparsedPara`, it may return a parse error, or one or more paragraphs (which will replace the `UnparsedPara`). Similarly, at the top level of a specification, an `UnparsedZSect` may become one or more sections if it can be parsed.

### 4.3 Predicates and Expressions

We shall not go into details about the structure of predicates and expressions etc., but will discuss some specific features and give a few short XML examples to give the flavour of our approach.

As for paragraphs, declarations and strokes, we define `Expr` and `Pred` to be abstract elements, and use substitution groups to allow specific concrete kinds of expressions and predicates to be used in their place. To capture the commonalities between various kinds of expressions, we define a hierarchy of XML types (Fig. 7). We expect that this same hierarchy can be used in Z tools that are written in object-oriented languages. Then the various concrete predicate and expression elements are defined as members of these types, as the following examples illustrate (`grp` stands for `substitutionGroup`):

```

<element name="OrPred"         type="Z:Pred2Type"   grp="Z:Pred"/>
<element name="ImpliesPred"    type="Z:Pred2Type"   grp="Z:Pred"/>
<element name="ForallPred"     type="Z:QntPredType"  grp="Z:Pred"/>
<element name="ExistsPred"     type="Z:QntPredType"  grp="Z:Pred"/>
<element name="FalsePred"      type="Z:FactType"     grp="Z:Pred"/>
<element name="TruePred"       type="Z:FactType"     grp="Z:Pred"/>

<element name="LambdaExpr"     type="Z:Qnt1ExprType" grp="Z:Expr"/>
<element name="MuExpr"         type="Z:QntExprType"  grp="Z:Expr"/>
<element name="LetExpr"        type="Z:Qnt1ExprType" grp="Z:Expr"/>
<element name="SetCompExpr"    type="Z:QntExprType"  grp="Z:Expr"/>

```

Some expressions and predicates have special features to enable the concrete syntax to be resurrected. Z has several conjunction operators ( $\wedge$ ,  $;$ , newline and the implicit conjunctions within  $a < b < c$ ), which are all represented by the `AndPred` element (of type `Pred2Type`) with an attribute to record which kind of conjunction it came from. The `RefExpr`, `App1Expr` and `MemPred` elements have a Boolean attribute called `Mixfix` to record whether the application uses mixfix notation or not.

*The Challenge of Nested Identical Names.* In Z it is quite common to have several levels of declarations nested inside one another. If two levels declare the same name  $X$ , then expressions inside the inner's scope cannot normally refer to the outer  $X$ . However, there are situations like the following example, where the instantiation of generic operators during type checking must introduce references to the outer  $X$  (the  $\#\{a\}$  becomes

TermType	supertype of all Z constructs
StrokeType	supertype of the 4 kinds of name decorations
AnnType	supertype of all annotations
TermAType	supertype of all annotatable constructs
Spec	
SectType	supertype of all section types
ZSectType	
UnparsedZSectType	
NarrSectType	
ParaType	supertype of all paragraph types
GivenParaType	
AxParaType	
FreeParaType	
ConjParaType	
OptempParaType	
UnparsedParaType	
DeclType	supertype of all declarations
VarDecl	
ConstDecl	
InclDecl	
PredType	
Pred2Type	supertype of all binary predicates
QntPredType	supertype of all quantifier predicates
FactPredType	supertype of the true/false predicates
ExprType	
Expr1Type	supertype of all unary expressions
Expr2Type	supertype of all binary expressions
LogExprType	supertype of all binary schema operators
QntExprType	supertype of all quantifier exprs
Qnt1ExprType	supertype of quantifier exprs with compulsory body
ExistsExprType	supertype of existential schema exprs
Expr0NType	supertype of exprs with 0 or more subexprs
Expr2NType	supertype of exprs with 2 or more subexprs
TypeType	supertype of all Z base types used in annotations
ParentType	
FreeTypeType	
BranchType	
SchTextType	
NameType	

**Fig. 7.** The hierarchy of XML complex types in ZML.

$\#[X]\{a\}$ ). This creates a problem, because naively introducing  $X$  at this point causes it to bind to the inner  $X$  rather than the outer  $X$ .

$[X]$

$$\frac{a : X}{\exists X : \mathbb{N} \bullet \#[a] = X}$$

None of the previous DTD or XML Schema proposals solve this problem. The traditional solution is to rename the bound  $X$ . But to allow exact resurrection of concrete syntax we do *not* want to rename bound variables. The Z standard solves this problem by creating suit-decorated synonyms of type names (e.g.,  $X\heartsuit$ ) and making implicit instantiations refer to those synonyms. We want a more general solution than this, so that tools can perform a variety of transformations, then produce correct XML using the original names, even though the scopes of those names may have changed.



CADiZ solves this problem by using references to link each name to a corresponding declaration. We do the same thing in XML, by using the ID and IDREF cross-reference features of XML to allow a variable reference to point to a specific variable declaration (which may not be the nearest nested name). Declarations of names may have an ID-valued attribute called `Id`, while references to names may have an IDREF-valued attribute called `Decl` which links to a declaration. Since soundness relies on following these references correctly, every Z tool must be capable of following them, and pretty printers must display the output unambiguously (either by renaming one of the bound variables, or by making the generic instantiations implicit again to hide the problem reference).

The full mark-up of the above example is shown in Appendix A. The XML for the declaration of the global name *X* is:

```
<GivenPara><DeclName Id="X.3"><Word>X</Word></DeclName></GivenPara>
```

while an expression that references this *X* can be marked up as:

```
<RefExpr><RefName Decl="X.3"><Word>X</Word></RefName></RefExpr>
```

## 5 Conclusions

We have defined an XML mark-up format for standard Z, based on combining the best features from the standard and several existing tools. The XML Schema has been validated, and several small examples have been validated against the schema. We are now seeking feedback and comments on the design, particularly on the following issues:

1. Two alternative approaches to annotations: the approach taken here is for each term to have an optional `Anns` slot that can contain arbitrary XML (which is not validated or checked in any way). An alternative approach would be to put new kinds of annotations into separate documents (with their own XML Schema) and use IDREF links to link each annotation to the appropriate Z term (which would have an ID attribute).
2. Two alternative approaches to narrative and non-standard portions of Z specification documents. Should narrative paragraphs and non-Z XML mark-up be viewed as subordinate to the Z, or should it be mixed in with the Z constructs on an equal basis (as in this paper)? The former approach allows stricter XML validation of the document, because every top-level paragraph is of a known type and can be checked (except that `Narrative` paragraphs would be allowed arbitrary contents). The latter approach (which we have taken) makes it easy to add new kinds of paragraphs (e.g., for Z extensions), even without extending the XML Schema, but means that standard Z tools will quietly ignore all unknown kinds of paragraphs.
3. Unparsed fragments. Is it really useful to be able to have some paragraphs or sections unparsed? Would an even finer granularity be useful (`Expr` and `Pred` etc.)? Or should we disallow unparsed portions and insist that this XML mark-up be used only for syntactically correct specifications?
4. Mathematical Symbols. We expect that the special symbols used in Z will normally be represented in XML documents using their binary Unicode representation (e.g., UTF8). However, this means that the documents are not ASCII-based and are only

human-readable if you have a full Unicode font. Would it be useful to define symbolic names for all the Z symbols (this can be done using DTD entities, or XML Schema elements with fixed contents) so that the Z specifications can be pure ASCII? Or will this be irrelevant once full Unicode fonts and Unicode editors become widely available?

Combining the best features from the Z standard and several existing tools has been worthwhile, as can be seen by considering the main influences on the XML structure. The `Specification` representation is influenced mainly by the form of XML. The `Section` representation is influenced mainly by Zeta. The `Paragraph` representation is influenced mainly by Standard Z, with the commonality between generics and non-generics taken from both `CADiZ` and Zeta, and the template representation in operator templates taken from Zeta. The `Predicate` representation is influenced mainly by `CADiZ` and Zeta, which use remarkably similar representations. The `Expression` representation falls between those of `CADiZ` and Zeta. The representations of schema text and names are influenced mainly by Zeta.

Next we plan to derive a set of open-source Java classes from this XML schema, preferably by using either JAXB<sup>5</sup> or XSLT to transform the schema into Java source. These Java classes will support the visitor design pattern [7], so that functionality such as type checkers, transformation tools, simplifiers and pretty printers can easily be written as add-on packages. This will dramatically reduce the usual initial barriers of creating new Z tools (parsing, type-checking etc.) and make it easier for student projects and other researchers to experiment with building new Z tools.

Another important step is for existing Z tools to support this XML format, by adding import and export functions that read and write it. `CADiZ` already exports an XML format that is close to this one.

## References

1. ISO/IEC 10646-1. *Information Technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane*. 2000.
2. ISO/IEC 10646-2. *Information Technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 2: Supplementary Planes*. 2001.
3. ISO/IEC 13568. *Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics*. 2002. First Edition 2002-07-01.
4. ISO 8879-1986. *Information Processing – Text and Office Systems – Standard Generalized Mark-up Language (SGML)*. ISO, 1986.
5. Nicholas Daley. Abstract syntax tree for Z. 591 Project Report, The Department of Computer Science, Waikato University, Hamilton, New Zealand, October 2002. Available from [marku@cs.waikato.ac.nz](mailto:marku@cs.waikato.ac.nz).
6. Jin Song Dong, Yuan Fang Li, Jing Sun, Jun Sun, and Hai Wang. XML-based static type checking and dynamic visualization for TCOZ. In *4th International Conference on Formal Engineering Methods*, pages 311–322. Springer-Verlag, October 2002.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, USA, 1995.

<sup>5</sup> See <http://java.sun.com/xml/jaxb>.

8. W. Grieskamp. ZETA. <http://uebb.cs.tu-berlin.de/zeta>, 2000.
9. E.R. Harold and W.S. Means. *XML in a Nutshell*. O'Reilly, 2001.
10. B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2), February 2000.
11. M. Saaltink. The Z/EVES system. In J. Bowen, M. Hinchey, and D. Till, editors, *Proc. 10th Int. Conf. on the Z Formal Method (ZUM)*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–88, Reading, UK, April 1997. Springer-Verlag, Berlin.
12. J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall International (UK) Ltd, second edition, 1992.
13. J. Sun, J.S. Dong, J. Liu, and H. Wang. An XML Schema for Z family. <http://nt-appn.comp.nus.edu.sg/fm/zml/zml.xsd>, 2001.
14. Jing Sun, Jin Song Dong, Jing Liu, and Hai Wang. A Formal Object Approach to the Design of ZML. *Annals of Software Engineering*, 13(1-4):329–356, June 2002.
15. I. Toyn. CADiZ. <http://www-users.cs.york.ac.uk/~ian/cadiz/>, 2001.
16. J. Wordsworth. An XML DTD for Z, October 1999.

## A XML Mark-Up of Example from Sect. 4.3

```

<NarrPara>
  <Content>First we declare X to be a given set.</Content>
</NarrPara>
<GivenPara>
  <DeclName Id="X.3"> <Word>X</Word> </DeclName>
</GivenPara>

<NarrPara>
  <Content>This axiomatic definition declares a:X, with the
    constraint:  $(\exists X:\mathbb{N} \ @ \ \# \{a\} = X)$ </Content>
</NarrPara>
<AxPara>
  <SchText>
    <VarDecl>
      <DeclName> <Word>a</Word> </DeclName>
      <RefExpr><RefName><Word>X</Word></RefName></RefExpr>
    </VarDecl>

    <ExistsPred>
      <SchText>
        <VarDecl>
          <DeclName> <Word>X</Word> </DeclName>
          <RefExpr><RefName><Word> $\mathbb{N}$ </Word></RefName></RefExpr>
        </VarDecl>
      </SchText>
      <MemPred>
        <TupleExpr>
          <AppExpr>
            <RefExpr>
              <RefName><Word>#</Word></RefName>
              <RefExpr>

```

```

        <RefName Decl="X.3"> <Word>X</Word>
      </RefName>
    </RefExpr>
  </RefExpr>
  <SetExpr>
    <RefExpr><RefName><Word>a</Word></RefName></RefExpr>
  </SetExpr>
</ApplExpr>
  <NumExpr Value="1"/>
</TupleExpr>
  <RefExpr><RefName><Word>=</Word></RefName></RefExpr>
</MemPred>
</ExistsPred>
</SchText>
</AxPara>

```