CZT: A Framework for Z Tools

Petra Malik and Mark Utting

The University of Waikato, Hamilton, New Zealand
{petra,marku}@cs.waikato.ac.nz

Abstract. The Community Z Tools (CZT) project is an open-source Java framework for building formal methods tools for Z and Z dialects. It also includes a set of tools for parsing, typechecking, transforming and printing standard Z specifications in LaTeX, Unicode or XML formats. This paper gives an overview of the CZT framework, including an introduction to its visitor design pattern that makes it possible to write new Z transformation tools in just a few lines of Java code. The paper also discusses several problems and challenges that arose when attempting to build tools based on the ISO Standard for Z.

1 Introduction

The Z specification language was adopted as an ISO standard in 2002 [1]. It can be used to precisely specify the behaviour of systems, and analyse it via proof, animation, test generation etc. However, one of the biggest barriers to the widespread use of the Z standard is the issue of tool support.

There are several industry-quality Z tools available that offer parsing and typechecking facilities (FuZZ[1], ZTC[2]) and some that also offer proof facilities (Z/EVES[3], CadiZ[4], ProofPower[5]). However, most of them do not support the ISO Standard for Z (CadiZ is the only one that supports almost all of the ISO Standard for Z). Furthermore, they use different versions of Z[6] or require different LaTeX macros, and there is little integration between the tools.

Other Z tools have been constructed as academic experiments or student projects, but these are typically not robust enough or complete enough for widespread use. Many good ideas and tools that were developed to prototype stage are no longer maintained or available now, because the project has finished or people have moved on.

[1] See `http://spivey.oriel.ox.ac.uk/mike/fuzz`.
[2] See `http://se.cs.depaul.edu/fm/ztc.html`.
[3] See `http://www.ora.on.ca/z-eves`.
[4] See `http://www-users.cs.york.ac.uk/~ian/cadiz`.
[5] See `http://www.lemma-one.com/ProofPower/index`.
[6] A quote from the ProofPower web page illustrates the challenge: *"The [ProofPower] ASCII mark-up is similar in spirit to the e-mail mark-up of the ISO Standard, but not in the details. There is no automatic way of transferring specifications between the ProofPower dialect of Z and other Z support tools at the moment. Now the standard has been finalised we hope that there will be more convergence."*

The Community Z Tools (CZT) project was proposed by Andrew Martin in 2001[7], with the goal of providing an open Internet-based community project that survives individuals, research projects, companies, etc. In 2003, the authors created the CZT project on Sourceforge[8], the world's largest open-source software development web-site.

Meanwhile Java core libraries for the ISO Standard for Z and some Z dialects have been designed and implemented. Alpha versions of several tools (parsers, printers, typechecker and a variety of transformers to and from different Z markups and other languages) are now available. People from all over the world have joined the project, and are either improving existing software or building new tools on top.

Figure 1 gives an overview of the CZT software, and provides an idea how it can be used. The XML Schema in the top left-hand corner of the diagram, which is introduced in Sect. 2, defines an XML file format for Z specifications. It can be used as an interchange format between different sessions and between external Z tools. In addition, the XML Schema has been used to generate Java interfaces and classes for annotated syntax trees (AST) for Z, as described in Sect. 3. These classes provide a convenient, markup-independent way to access the syntactic objects of a Z specification. Most of the tools provided either create, modify, or traverse an AST. All the tools in Fig. 1 that take an AST as input use our *CZT visitor* design pattern described in Sect. 4 to traverse the AST.

The CZT parser can transform Z specifications in various formats into an AST representation. Once available as an AST, a wealth of options for further processing are possible. Using the XML file format, it can be passed on to other tools. Alternatively, the specification can be typechecked, animated, or translated into a variety of formats such as, for instance, B, HTML, JML, or another markup language. Section 5 gives an overview of the CZT tools available.

In Sect. 6, the paper discusses some problems that arose while designing and implementing CZT. Finally, Sect. 7 contains some concluding remarks.

2 ZML—XML Markup for Z

The heart of CZT is an XML Schema that describes the XML markup for Z specifications (ZML). It is an interchange format that can be used to exchange parsed and even typechecked specifications between sessions and tools. It was first described in [2] and has been enhanced since then. To allow for evolution of the ZML format, version numbers have been introduced. Each ZML specification should specify which version of the ZML format it is using. The original proposal has version number 1.0, which is also the default when no version number is given. At the time of writing, the current version number is 1.3. A list of changes can be found at the web-site `http://czt.sourceforge.net/zml`.

ZML was designed in a way to capture sufficient information to rebuild the constructs from which it was originally parsed and, at the same time, to minimise

[7] See `http://web.comlab.ox.ac.uk/oucl/work/andrew.martin/CZT`.

[8] See `http://czt.sourceforge.net`

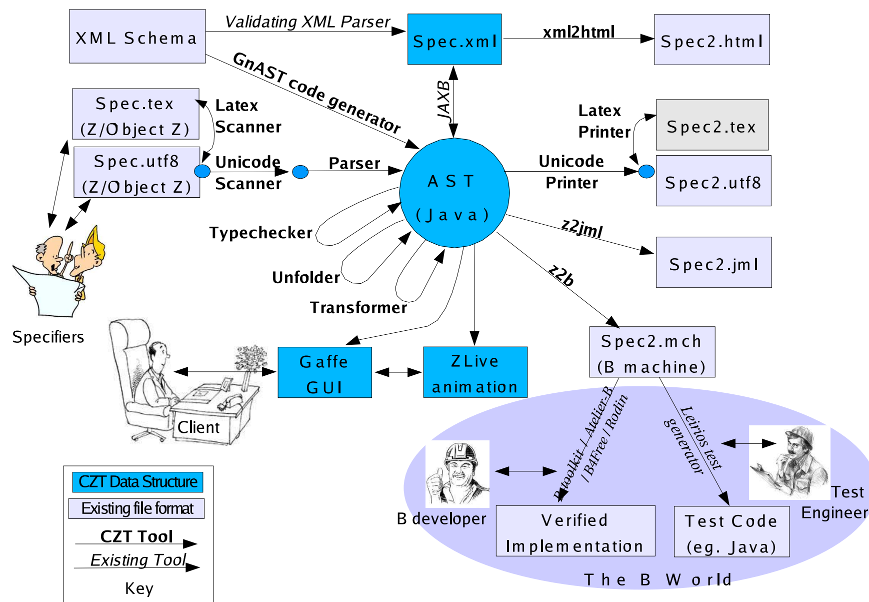

Fig. 1. Overview of the CZT architecture.

the number of cases a tool needs to deal with. This has been achieved by adding a type hierarchy—parts of it are shown in Fig. 2—to reflect commonalities, and by using common XML tags for similar constructs. Attributes are then used to distinguish between the similar constructs.

A ZML snippet for the Z schema given by the following LaTeX markup

```
\begin{schema}{BirthdayBook}...\end{schema}
```

can be seen in Fig. 3. As described in the ISO Standard for Z, a schema definition is semantically equivalent to an axiomatic description associating the name of the schema with the schema text. In ZML, the `AxPara` is used to represent axiomatic descriptions, generic axiomatic descriptions, schema definitions, generic schema definitions, horizontal definitions, generic horizontal definitions, and generic operator definitions. The attribute `Box` distinguishes between the different types of paragraphs and ensures that the original construct can be restored. `SchBox` is short for schema box and indicates that we are dealing with a schema definition.

The XML Schema definition for `AxPara` is given in Fig. 4. The `AxPara` element defines the XML tag `AxPara`. The attribute `substitutionGroup` is used to define the inheritance hierarchy (see Fig. 2). `AxPara` is in the substitution group of `Para`, that is, it is a special paragraph. The complex type `AxPara` is used to define the children and attributes of the element `AxPara`. The `AxPara` element contains, in addition to the elements defined in `Para`, a possibly empty list of generic parameters (`DeclName`) followed by a schema text (`SchText`).

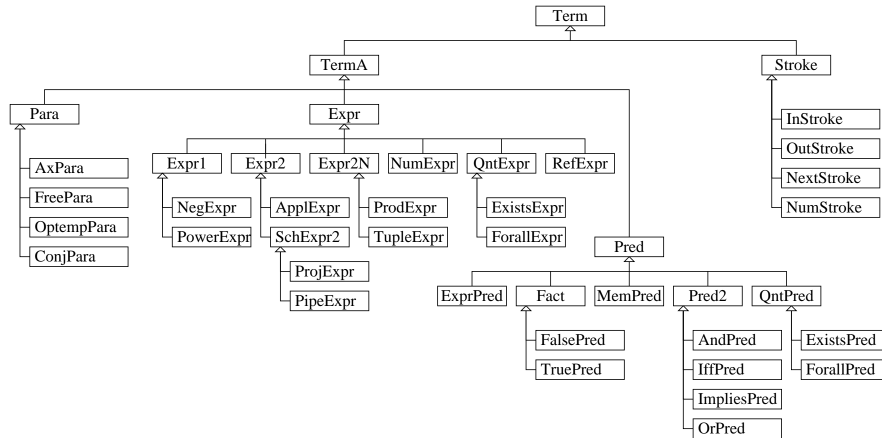

Fig. 2. Part of the inheritance hierarchy.

```
<AxPara Box="SchBox">
  <SchText>
    <ConstDecl>
      <DeclName><Word>BirthdayBook</Word></DeclName>
      <SchExpr>...</SchExpr>
    </ConstDecl>
  </SchText>
</AxPara>
```

Fig. 3. ZML snippet.

3 AST—Annotated Syntax Tree

The *annotated syntax tree* (AST) provides a tree view of a parsed Z specification using Java interfaces and classes. This allows easy access to syntactical objects like, for instance, paragraphs, predicates, and expressions, from within Java programs. Currently, CZT contains AST interfaces and classes for Z [1], Object Z [3], and TCOZ [4]. Support for other extensions is conceivable.

The AST interfaces reflect the XML markup defined in ZML very closely. In fact, the AST interfaces and classes were automatically generated from the XML Schema describing ZML using our code generator *GnAST* (GeNerator for AST). For each element defined in the XML Schema, a corresponding interface and implementing class is generated. A user of the AST should always use references to the interfaces instead of references to concrete classes. This protects the user from changes to the underlying implementation and allows the use of different implementations of the AST interfaces.

The *abstract factory pattern* [5] is used to provide a way to create AST objects without any knowledge of their concrete classes. For example, a Z tool that creates a new `AxPara` object should *never* call the constructor of an `AxPara` implementation, instead it should use `factory.createAxPara()`, where `factory` is an interface that has methods for creating each kind of AST node. Different

```
<xs:element name="AxPara" type="Z:AxPara" substitutionGroup="Z:Para">
  <xs:annotation>
    <xs:documentation>
      A (generic) axiomatic paragraph, (generic) schema definition,
      or (generic) horizontal definition.
    </xs:documentation>
  </xs:annotation>
</xs:element>

<xs:complexType name="AxPara">
  <xs:complexContent>
    <xs:extension base="Z:Para">
      <xs:sequence>
        <xs:element ref="Z:DeclName"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="Z:SchText"/>
      </xs:sequence>
      <xs:attribute name="Box" type="Z:Box"
          use="optional" default="AxBox"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Fig. 4. XML Schema definition for `AxPara`.

implementations of this interface can then be used to create instances of different concrete AST classes. By passing the tool an alternative factory object, one can control which implementations of the AST interfaces it will create. The CZT software provides a standard factory that creates instances of the implementation classes supplied within CZT. A user is free to use their own classes that implement the AST interfaces by providing an alternative factory.

The inheritance hierarchy of AST interfaces and classes—parts of it are shown in Fig. 2—is the same as the element inheritance hierarchy defined in the XML Schema. The leaves of this tree represent concrete classes, which can be instantiated via the factory mentioned above, while the inner nodes represent abstract classes, which cannot be directly instantiated.

For each child and attribute defined for a ZML element, the corresponding Java interface provides getter and setter methods. For example, the Java interface `AxPara` given in Fig. 5 is automatically generated from the `AxPara` element definition given in Fig. 4, which is taken from the XML Schema that describes ZML. The `AxPara` element is defined to extend `Para` and to contain, in addition to the elements defined in `Para`, a possibly empty list of declaring names (`DeclName`) followed by a schema text (`SchText`). There is also an attribute called `Box`.

Accordingly, the `AxPara` interface extends `Para` and contains, in addition to the methods defined in the interface `Para`, the following getter methods: `getDeclName` returning a list, `getSchText` returning a schema text (`SchText`), and `getBox` returning a `Box`, which is an enumeration used to indicate whether this was parsed from a schema text, axiomatic definition, or horizontal definition. In addition to getter methods, setter methods are provided. Note that there is no setter method for `DeclName` since the getter returns a list, and the `List` interface can be used to add and remove elements.

```
/**
 * A (generic) axiomatic paragraph, (generic) schema definition,
 * or (generic) horizontal definition.
 *
 * @author GnAST code generator
 */
public interface AxPara extends Para
{
  /** Returns a list of declaring names (formal parameters). */
  ListTerm getDeclName();

  /** Returns the schema text. */
  SchText getSchText();

  /** Sets the schema text. */
  void setSchText(SchText schText);

  /** Returns the Box attribute. */
  Box getBox();

  /** Sets the Box attribute. */
  void setBox(Box box);
}
```

Fig. 5. AST interface for `AxPara`.

The getter and setter methods defined by the AST interfaces provide a convenient way to access and manipulate individual AST objects. However, it is often necessary to manipulate a whole AST in a certain way. The following section shows how complete trees can be handled with just a few lines of code.

4 The CZT Visitor Design Pattern

The visitor design pattern [5] provides a way to separate the structure of a set of objects from the operations performed on these objects. This allows a new operation to be defined without modifying the AST classes. To define a new operation, all you need to do is to implement a new visitor class. An example of a visitor class is given in Fig. 6. It traverses an AST and prints the word-parts of all declaring names (`DeclName`).

A variety of variants [6] of the visitor design pattern have been proposed, all having different advantages and disadvantages. The visitor design described in this section is the result of a lot of experimentation and redesign. We discussed many variants and implemented three or four different designs before we found a design that met all our requirements.

The standard visitor pattern described in [5] is based on a double dispatch mechanism. The set of objects on which new operations are to be defined must support this pattern by providing an `accept(Visitor v)` method. The `Visitor` is an interface that defines `visit` methods for each class of object that is to be visited (like `visitDeclName` in Fig. 6). The `accept` method of an object calls back the correct `visit` method for its class. This allows different implementations of the `Visitor` interface to perform different operations on the objects.

The disadvantage of this approach is that it is difficult to add new AST classes, because each new AST class implies that a new method needs to be added

```
import net.sourceforge.czt.base.ast.Term;
import net.sourceforge.czt.base.visitor.TermVisitor;
import net.sourceforge.czt.base.visitor.VisitorUtils;
import net.sourceforge.czt.z.ast.DeclName;
import net.sourceforge.czt.z.visitor.DeclNameVisitor;

public class DeclNamePrintVisitor
  implements TermVisitor, DeclNameVisitor
{
  public Object visitTerm(Term term)
  {
    VisitorUtils.visitTerm(this, term);
    return null;
  }

  public Object visitDeclName(DeclName declName)
  {
    System.out.println(declName.getWord());
    return null;
  }
}
```

Fig. 6. A simple visitor.

to the `Visitor` interface, which in turn requires modifications to all its existing implementations. Within CZT, we want to support extensions like Object Z, TCOZ, etc, and therefore need to be able to easily extend the AST classes. Another disadvantage is that each visitor class needs to implement a fixed set of methods, one for each AST class—and there are a large number of AST classes defined in CZT. While it is possible to provide default `Visitor` implementations, then use inheritance to override just the desired methods, the lack of multiple inheritance in Java often makes this a clumsy or undesirable solution.

The CZT visitor incorporates the advantages of the *acyclic visitor* [7] pattern and the *default visitor* [8] pattern, as well as some new twists. The *acyclic visitor* pattern allows new AST classes to be added without changing the existing visitor classes. This is done by defining a visitor interface for each object and using a dynamic cast in the AST `accept` methods. The *default visitor* pattern adds another level of inheritance to the visitor pattern, making it possible to implement default behaviour by taking advantage of the AST inheritance relationships. That is, the visitor classes of the default visitor pattern define a `visit`AAA method for each *abstract* AST class AAA, in addition to the usual `visit`CCC methods for each concrete AST class CCC. Visitors can then define default behaviour within these extra (abstract) `visit` methods. If not overridden, a concrete `visit`CCC method typically just calls the `visit`AAA method which corresponds to its closest superclass.

```
public interface AxParaVisitor extends Visitor
{
 /** Visits an AxPara. */
 Object visitAxPara(AxPara axPara);
}
```

Fig. 7. The AxParaVisitor interface.

Now we describe the CZT visitor pattern. In CZT, a visitor interface is defined for every AST class, including abstract superclasses. As an example, the visitor interface for `AxPara` is given in Fig. 7. If a visitor implements this interface, then any `AxPara` AST nodes that it visits will call the visitor's `visitAxPara` method. However, if the visitor does not implement the `AxParaVisitor` interface, then the `AxPara` AST nodes will search up though their superclasses and call the first `visit`*AAA* method that the visitor implements (for example, `visitPara` or `visitTermA` or `visitTerm`). Figure 8 illustrates how the `accept` method for `AxPara` implements this semantics. With this approach, the AST classes themselves take care of calling the closest (with respect to the inheritance hierarchy) `visit` method implemented by the visitor.

```
public class AxParaImpl extends ParaImpl implements AxPara
{
  ...

  /** Accepts a visitor. */
  public Object accept(Visitor visitor)
  {
    if (visitor instanceof AxParaVisitor) {
      AxParaVisitor axParaVisitor = (AxParaVisitor) visitor;
      return axParaVisitor.visitAxPara(this);
    }
    return super.accept(v);
  }
  ...
}
```

Fig. 8. The `accept` method of AxPara.

```
public interface Term
{
  /** Accepts a visitor. */
  Object accept(Visitor visitor);

  /** Returns an array of all the children of this term. */
  Object[] getChildren();

  /** Creates a new object of the implementing class
   *  with the objects in args as its children. */
  Term create(Object[] args);
}
```

Fig. 9. The `Term` interface.

The `Term` interface given in Fig. 9 is the base of all AST objects and must therefore be implemented by every AST class. The two additional methods defined in the `Term` interface provide a convenient way to handle AST classes generically within visitors. The `getChildren` method provides a generic alternative to the getter methods by returning all children of a term as an array. This allows us to write a single `visitTerm` method that recurses through the

```
import net.sourceforge.czt.base.ast.Term;
import net.sourceforge.czt.base.visitor.TermVisitor;
import net.sourceforge.czt.z.ast.DeclName;
import net.sourceforge.czt.z.util.Factory;
import net.sourceforge.czt.z.visitor.DeclNameVisitor;

/** A visitor that copies a given AST (except for annotations)
 *  into one where all strokes are removed from each DeclName.
 */
public class StrokeKiller
  implements TermVisitor, DeclNameVisitor
{
  private Factory factory_ = new Factory();

  public StrokeKiller()
  {
  }

  public StrokeKiller(Factory factory)
  {
    factory_ = factory;
  }

  public Object visitTerm(Term term)
  {
    Object[] args = term.getChildren();
    for (int i = 0; i < args.length; i++) {
      if (args[i] instanceof Term) {
        args[i] = ((Term) args[i]).accept(this);
      }
    }
    return term.create(args);
  }

  public Object visitDeclName(DeclName declName)
  {
    return factory_.createDeclName(declName.getWord(), null);
  }
}
```

Fig. 10. Another simple visitor.

entire AST tree (see `visitTerm` in Fig. 10). This default `visitTerm` method is so common that it is supplied in the `VisitorUtils` library class, which has been used to implement the `visitTerm` method in Fig. 6.

Similarly, the `create` method is a convenient way for default `visit` methods to change the contents of a tree node, while retaining its original type. The `create` method is similar to a clone, but allows new children to be provided. These children are typically returned by the `visit` calls to the original children. Figure 10 shows a visitor that copies an AST into one where all decorations, i.e. strokes, are removed from `DeclName` elements.[9]

The visitor needs to traverse the tree to find all `DeclName` objects. This traversal is handled in the `visitTerm` method, which is called for all AST classes except `DeclName`. It makes sure that all children are visited. Furthermore, the results of visiting the children are used to create a new object of the same

[9] Note that annotations are not copied. If we wanted to retain annotations as well, the `TermAVisitor` interface could be implemented in a way that also copies annotations.

type containing the new children. When a `DeclName` accepts this visitor, the `visitDeclName` method is called. In this case, a new `DeclName` is created with the same name as the one that is visited, but no decorations are added.

This visitor demonstrates the use of the `getChildren` and `create` methods, as well as the use of a factory to create new `DeclName` objects. Note that this visitor has no reference to concrete implementations of the AST interfaces; only references to AST interfaces are used. The visitor uses the standard factory provided within CZT if no factory is given. It can also be configured to use alternative factory implementations.

Visitors are extensively used throughout CZT. Virtually all of the tools that access or manipulate an AST, like the typechecker, printers, etc., are visitors or use visitors to achieve their functionality. The advantage of the CZT visitors is that the amount of code that needs to be written is directly proportional to the AST nodes that need to be transformed or accessed—recursion through all the other AST nodes is done by the default `visitTerm` method. This makes it easy to write visitors that transform Z in some new way. It is simple to combine such a visitor with the existing Z parsers and printers in CZT, to quickly obtain a new Z tool.

5 CZT Tools

The CZT software also includes a set of tools as shown on the diagram in Fig. 1. At the time of writing, the scanners, parsers, printers and typechecker are quite robust and well-tested.[10]

Figure 11 gives an overview of the different parser components. The components responsible for parsing specifications in Unicode are given on the right side. As described in the ISO Standard for Z and in [9], scanning and parsing is performed in several steps. The first step is the scanning phase carried out by the `Unicode Scanner`. In fact, the scanner itself consists of several components: the `Context-Free Scanner`, the `Keyword Scanner`, the `Smart Scanner`, and the `Operator Scanner`.

The `Context-Free Scanner` is an implementation of the context-free lexis described in the ISO Standard for Z [1, §7.2]. JFlex[11], a Java scanner generator, is used to generate this class. The context-free lexis is followed by the context-sensitive lexis implemented by the `Keyword` and `Operator Scanner`. The `Smart Scanner` resolves one of the context-sensitive ambiguities in the Z grammar that is discussed in the ISO Standard for Z [1, §8.4, Note 4]. For example, in $\{x, y, z ...\}$, if the x, y, z is followed by ':', then it is part of a declaration (a set comprehension) and declares new names. Otherwise it is a set extension, and x, y, z must already have been declared somewhere else. The `Smart Scanner` performs lookahead to resolve this ambiguity.

[10] The typechecker is the newest addition and has checked only about 2000 lines of Z so far.

[11] See `http://www.jflex.de`.

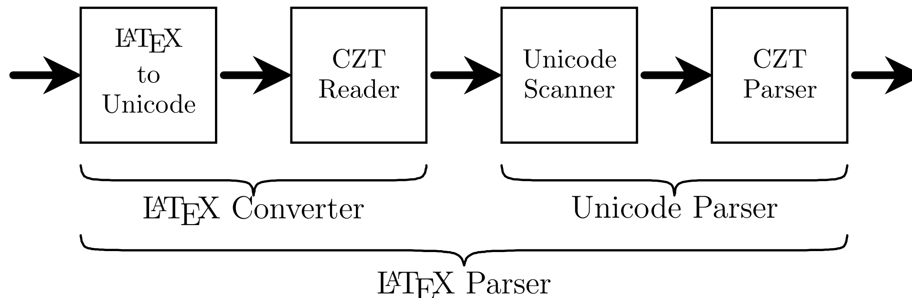

Fig. 11. The parser architecture.

The scanning phase is followed by the parsing phase. The `CZT Parser` is a look-ahead LR parser (LALR parser) generated by the Cup[12] Java parser generator. See Sect. 6 for a description of some problems that we encountered while writing this parser.

In order to parse a specification in LaTeX markup, it is first converted into Unicode using the LaTeX markup to `Unicode Converter` shown on the left side, and subsequently parsed by the `Unicode Parser` described above. One requirement on the parser is that it retains the original line and column positions from the file that is parsed. This location information is added to the AST nodes as `LocAnn` annotations, so that tools can later provide error messages that contain accurate location information pointing to the original specification document. It is difficult to preserve this location information when a specification in LaTeX markup is converted into Unicode before it is parsed since we want to preserve the location within the LaTeX document, not within the temporary Unicode document.

Our solution to this problem is to design the LaTeX markup to Unicode converter so that it returns a token stream (with line and column number information) rather than just a sequence of Unicode characters. Then a special reader, called `CZT Reader`, accepts these tokens and passes the Unicode characters into the `Unicode Parser`. In addition to the usual methods provided by the Java class `Reader`, the `CZT Reader` class provides special methods to access line and column number information. Those methods are then used by the `Unicode Scanner` to obtain location information for the tokens. This is an elegant way of retaining the original LaTeX line and column positions, even though the LaTeX input briefly becomes Unicode as described in the ISO Standard for Z. This solution is far simpler than writing a completely separate set of scanners specifically for LaTeX input.

In addition to a parser for Z, an Object Z parser is provided, and a TCOZ parser is under development. The Object Z parser and scanner extend the Z parser and scanner by some more tokens and grammar rules. Unfortunately, it is quite difficult to reuse code from an automatically generated scanner or parser,

[12] See `http://www.cs.princeton.edu/~appel/modern/java/CUP`.

and Cup does not provide means to do so. To avoid duplicated code, XML templates are used that contain the different parser and scanner variants. From this, the different input files for JFlex and Cup are generated. This maximises the commonality between the parsers and minimises versioning and maintenance problems.

A parsed Z specification can be typechecked using the `CZT Typechecker`. The type inference rules are defined in the ISO Standard for Z [1, §13] and explained in [10]. The `CZT Typechecker` is implemented as an AST visitor. It visits every term in the tree to determine its type (if any) and detects typing errors. If there are no typing errors in an expression, a `TypeAnn` annotation is added to the term, recording the expression's type. If there are type errors, a `TypeErrorAnn` annotation is added to the term. Each `TypeErrorAnn` records the position of the typing error in the original specification document using `LocAnn` annotations, and contains an error message describing the problem.

The typechecker also maintains a list of references to all `TypeErrorAnn` instances for easy printing of error messages. As defined in the ISO Standard for Z [1, §10.2], the typechecker also adds an annotation containing the signature of a paragraph (the empty signature for paragraphs that contain no global declarations), as well as a `SectTypeEnvAnn` annotation, which records the global declarations and their types visible in this section (including declarations for parent sections).

The `Unfolder` tool (which unfolds schema calculus operators and some other Z constructs) and the `Transformer` tool (which transforms ASTs by applying user-defined rewrite rules) are still under development. The `zml2html`, `z2jml` and `ZLive` animator tools are preliminary prototypes only. The `z2b` tool handles the Birthday Book example [11], but does not yet translate all Z constructs. The aim behind developing this Z-to-B translator is to give Z users access to the excellent refinement tools available in the B world (such as the B-Toolkit[13], Atelier-B[14] and the forthcoming Rodin tools[15]) and also to the BZ-TT automatic test generation tools [12] for B, which are being released commercially.[16]

6 Challenges and Problems

This section describes some of the difficulties that we have encountered in trying to build a 'strongly conforming' Z tool as defined in the ISO Standard for Z [1, §5.2.5] (one that accepts all correct Z specifications, and rejects all incorrect specifications). We describe the solutions that we have adopted, and make some suggestions for how the ISO Standard for Z could be changed in the future to avoid some of the difficulties.

[13] See `http://www.b-core.com`.
[14] See `http://www.atelierb.societe.com/index_uk.html`.
[15] See `http://rodin-b-sharp.sourceforge.net`.
[16] See `http://www.leirios.com`.

6.1 Grammar Complexities

The grammar in the ISO Standard for Z is intentionally ambiguous, and needs significant modification before it can be used as the basis for a parser. For example, there is ambiguity between schemas as expressions (sets of bindings) and predicates [1, §8.4]. Fortunately, Ian Toyn showed us some elegant strategies for obtaining an LALR(1) grammar[17] by merging expressions and predicates, thus parsing a larger set of inputs than desired. We use Java runtime typing (`instanceof`) to 'fix up' the types of the AST after parsing. That is, inputs that are definitely predicates are parsed into `Pred` objects, inputs that are definitely expressions are parsed into `Expr` objects, and ambiguous inputs, which could be interpreted as either expressions or predicates, are also initially parsed as `Expr` objects, because expressions are a subset of predicates in the ISO Standard for Z. If one of these ambiguous `Expr` objects is later placed into the AST in a context which requires it to be a predicate, it is then converted into a `Pred` object by wrapping it in an `ExprPred` object.[18] Conversely, if we have an AST context that requires an expression, but we find that the parsed subexpression is actually an instance of `Pred`, we throw an exception to report a syntax error.

6.2 Multiple Passes

The Z standard allows Z sections, operator definitions and LaTeX markup definitions to appear in almost any order. For example, operators can be used before they are defined. This means that, in general, it is necessary to make several passes over the input. In this subsection, we describe what passes are necessary and which ones we have currently implemented.

Firstly, the Z standard states that a Z specification is comprised of a sequence of Z sections, but suggests that tools should not require sections to appear in order. That is, tools should allow a section to appear prior to one of its parent sections. The Z standard does not define any relationship between *files* and section or specifications. In CZT, we allow a specification to be split over several files, and allow each file to contain one or more sections. The *parent* dependencies between sections means that it may be necessary to reorder sections *within* a file, as well as handle dependencies *between* files. The Z standard does not allow circular dependencies between sections. The CZT tools do not allow circular dependencies between files either.

Secondly, the scope of an operator template includes the whole section in which it appears. That is, a user defined operator can be used before its definition, making it impossible to parse a specification without reordering the paragraphs so that operator definitions are parsed before all other paragraphs. Thirdly, the same scope problem arises with LaTeX markup directives; a user

[17] The 1 means one token of lookahead. But note that this is *after* the smart scanner has done unbounded lookahead to resolve the syntactic ambiguities described in [1, §8.3, Note 4].

[18] The CZT AST hierarchy currently separates expressions and predicates, as shown in Fig. 2, for clarity of programming and to obtain stronger typechecking in Java.

defined LaTeX command might be used before its rendering information is given via a markup directive.

Thus in the general case, it seems necessary to make five character-level passes over the input file of a LaTeX Z specification (while retaining the line and column positions of the original source file!):

1. to reorder the sections into parent-before-child order,
2. to collect the LaTeX markup directives within each section,
3. to convert the LaTeX input into Unicode characters,
4. to parse the operator template definitions,
5. to parse the Unicode and build the AST.

To simplify this, the CZT tools currently assume that sections are already in the correct order within each file and that LaTeX markup directives and operator templates appear before they are used. This is the case in most specifications. It even seems desirable for human readability that operators are defined before they are used. These restrictions mean that pass 1 is unnecessary and allows the CZT tools to perform all the remaining passes within a single pipeline, which is simple and efficient. However, we intend to provide alternative modes and tools that will perform the reordering and additional passes when the user desires.

6.3 Operator Templates

Z provides a way to define new operators using so called *operator templates*. For example, the operator template

$$\mathrm{generic}\ 30\ \mathrm{leftassoc}\ \ (_\ a\ _\ b\ _)$$

defines a ternary operator with *name* "$_\ a\ _\ b\ _$", with *precedence* 30, and *associativity* left. The ISO Standard for Z [1, §8.3] allows the same *word* to appear in several different operator templates, but with quite complex restrictions on which kinds of reuse are 'acceptable'. For instance, the operator template

$$\mathrm{generic}\ 30\ \mathrm{leftassoc}\ \ (_\ a\ _\ c\ _)$$

reuses the word "a" acceptably, but

$$\mathrm{generic}\ 40\ \mathrm{leftassoc}\ \ (_\ a\ _\ c\ _)$$

would be in conflict because both templates use the word "a" and have different precedence.

Each operator template also defines an association between operator words and so-called *operator tokens* [1, §7.4.4]. The context-sensitive lexis makes sure that operator names are lexed as the corresponding operator tokens. This implies that the association between word and token must be a function.

Thus, a Z tool needs to check for each Z section that its templates

– that share the same word have the same precedence,

– that share the same precedence have the same associativity, and
– that share the same word associate it with the same operator token.

When a section inherits multiple parent sections, the tool must detect any inconsistencies between the parents. Although these rules are complex, they seem necessary to prevent users defining ambiguous sets of operators. The CZT parser maintains the following data structures for each Z section to check these rules:[19]

$[OpName, OpWord, OpToken, OpTemplate]$

$Assoc ::= leftAssoc \mid rightAssoc$

OpTable

$operators : OpName \pfun OpTemplate$
$operatorWords : OpWord \pfun OpToken$
$operatorPrecedence : OpWord \pfun \mathbb{N}$
$associativity : \mathbb{N} \pfun Assoc$

The ISO Standard for Z does not answer some questions about user-defined operator precedences. For example, it does not specify what range of precedence numbers is allowed, which presumably means that any non-negative integer is allowed. This means that all 'strongly conforming' Z tools must accept an operator template definition whose precedence numeral is one million digits long. This seems unnecessarily complex. To improve interoperability between Z tools, and to make it easier to build strongly conforming tools, we recommend that future versions of the ISO Standard for Z specify a fixed range of operator precedences, such as $0 \mathrel{..} 1000$ (the CZT tools currently allow any integer in the range $0 \mathrel{..} 2^{31} - 1$).

Allowing user-defined operators with thousands of precedence levels makes parsing more difficult, as noted in the ISO Standard for Z. A fixed LL(k) or LALR(k) grammar is not sufficient, instead we parse all user-defined operators at the same precedence level, then add a post-processing phase that rotates adjacent levels of the AST to respect the user-defined precedences. The transformation rules for this post-processing are described in the ISO Standard for Z [1, §8.3, Note 3]. For example:

$$(e_1 \ infix_1 \ e_2) \ infix_2 \ e3 \implies e_1 \ infix_1 \ (e_2 \ infix_2 \ e_3)$$

It was not initially clear to us whether these rules apply only to binary infix operators (like "$_ + _$"), or to all infix operators (for example, a user could define the ternary operator "$_ \clubsuit _ \spadesuit _$"). It turns out that the latter is the case. Similarly, the prefix and postfix operators mentioned in these rules may

[19] This little Z specification has been checked with the CZT parser and typechecker. Our first try gave a syntax error on the 'leftassoc' word because it is a reserved word in standard Z! After we renamed it to 'leftAssoc' (similarly for 'rightAssoc'), it parsed and typechecked correctly.

be operators with more than one argument, like the postfix relational image operator “$_ (\!| _ |\!)$”.

Overall, operator templates are one of the most complex aspects of the ISO Standard for Z (they account for over 50% of the Z grammar!), but give Z users enormous flexibility in defining new mathematical notation.

6.4 Newlines and White Space

There are some unclear issues related to which Unicode characters should be used for line breaks, paragraph breaks etc. These have been addressed in a “Draft Technical Corrigendum 1” [13] to the ISO Standard for Z. This says that the NLCHAR mentioned throughout the ISO Standard for Z is in fact the Unicode character `U+2028` (Line Separator). However, in practice, Unicode files are likely to continue using the platform-specific line terminator characters (`LF` for Linux, `CRLF` for Windows etc.) and the algorithm for translating these into Z characters is left somewhat vague, stating only that it should follow the Unicode standard.[20] In CZT, we currently treat all line termination sequences (`LF`, `CR`, `CRLF`) as equivalent to the Unicode Line Separator (`U+2028`) which is NLCHAR in the ISO Standard for Z.

Even after one has decided which Unicode characters correspond to newlines, the handling of newlines is very complex. It is described in the ISO Standard for Z [1, §7.5] where *newline categories* for tokens are defined. The newline category of a newline depends on the newline categories of the closest tokens. However, it also depends on the context where it appears. Quoting from the ISO Standard for Z: “*All newlines are soft outside of a `DeclPart` or a `Predicate`*”. But newlines are supposed to be handled in the scanning phase where nothing is known about grammatical objects like `DeclPart` or `Predicate`. Furthermore, it is not clear to us what “outside” actually means. Is a newline after the formal parameters of a schema definition (which is just before the `DeclPart`) *outside* the `DeclPart` or not? According to the newline categories rule, a newline between formal parameters and declaration would be hard and therefore result in a parse error. Since it makes sense to allow newlines at this position, we conclude that it is outside the `DeclPart` and therefore soft. To handle this case, we had to modify the grammar of the parser to allow newlines there.

The handling of white space is very complex and fragile, particularly when translating to and from LaTeX markup. White space is meaningful in Unicode (“`x’`” is different from “`x ’`”—and see [1, §8.4, Example 1] for a lovely example!), but not in the LaTeX markup, where explicit spacing commands must be used instead. The LaTeX input “`f a`” meant to be an application of function `f` to `a` is interpreted as the single word “`fa`”, which usually results in interesting type errors. It must be written as “`f~a`” or using any other spacing command. A more tricky example is the following: the LaTeX inputs “`[a_1, b_1]`” and “`[a_1 , b_1]`” are both wrong. They result in parse errors since “`a_1,`” is

[20] See `http://www.unicode.org/reports/tr13/tr13-9.html` for an overview of the issue.

treated as one word (note the comma at the end). The reason for this is that the corresponding Unicode representation of “`a_1`” ends with a “word glue” character that glues the following comma on the word. It must therefore be written as “`[a_1~, b_1]`”. To ease the problem, the CZT parser makes use of the nice idea from FuZZ to print a warning whenever it finds a suspicious word like, for instance, one that contains spaces or newlines in its LaTeX markup.

6.5 Retaining LaTeX Markup Directives

LaTeX markup directives contain rendering information and specify the conversion of user defined LaTeX commands to the corresponding sequences of Z characters [1, §A.2.3]. A typical directive is, for example,

```
%%Zchar \Delta U+0394
```

stating that the LaTeX command “`\Delta`” is rendered as the Unicode character with code point U+0394. When converting LaTeX markup to Unicode, this information is used but not retained in the resulting Unicode markup. This is problematic since the scope of a markup directive is the section in which it appears and any sections of which it is an ancestor. This implies that the parents of a section written in LaTeX markup should also be given in LaTeX markup.

ZML is intended to retain as much of the layout of the original specification as possible. If a specification is given in LaTeX markup, we want to retain the LaTeX markup directives in ZML. This also makes it possible for a LaTeX Z section to have a ZML section as a parent. So we extended ZML by a new kind of paragraph, `LatexMarkupPara`, which contains LaTeX markup directives. These paragraphs are added directly to the AST by the LaTeX to Unicode converter (the parser does not see them, since they do not appear in the Unicode markup).

6.6 Unicode to LaTeX Conversion

The translation from Unicode to LaTeX markup turned out to be more difficult than we expected. The ISO Standard for Z describes scanning and parsing of Z specifications on the basis of Unicode markup. In Annex A, LaTeX and e-mail markup are introduced and its conversion to Unicode is described. However, no algorithm for the translation of Unicode to LaTeX or e-mail markup is given.

Initially we thought that a simple character or token translater would be sufficient for translating specifications in Unicode into LaTeX markup. However, the task turned out to be quite difficult and a kind of parser is required to do this task. The reason for this is that the Unicode character ‘|’ (U+007C), must be translated into different LaTeX tokens depending on the context where it appears. In top-level paragraphs like axiomatic definitions and vertical schema definitions it must be translated into “`\where`” see [1, §A.2.7], but within predicates and expressions (including horizontal schema definitions) it is left as the character ‘`|`’. In future versions of the ISO Standard for Z, it might be nice to use separate Unicode characters to represent these rather different separators. For example,

'├' (U+251C Box Drawings Light Vertical and Right) would look nicer as the separator within top-level paragraphs than '|'.

CZT provides a LaTeX printer for parsed specifications. Thus, in order to convert Unicode to LaTeX markup, a specification must be parsed before it can be printed as LaTeX markup. This means that only syntactically correct specifications can be converted. CZT also provides a token converter from Unicode to LaTeX markup that does not require parsing, but is affected by the '|' problem described above. The current workaround is to translate the Unicode character '|' (U+007C) to "\where" when '|' is on a line on its own, otherwise it is translated to '|'. Thus, the converter works only for specifications that obey this formatting rule.

6.7 Summary

Overall, we conclude that, partly for historical reasons, Z is a very complex language to scan and parse. It took us about 1 person-year to implement the scanners and parsers (Tim Miller developed most of the current Z and Object-Z parsers), with many iterations of finding increasingly complex syntax examples that required redesign of our architecture. If it was not for the fact that both authors of this paper are perfectionists and enjoy the challenge of getting every last case to work (elegantly), we probably would not have finished. In spite of these difficulties and the above comments, we note that the ISO Standard for Z does an excellent job of describing most of the complexities. Some very important implementation considerations are just briefly mentioned in notes and examples, but it is obvious that a great deal of care was taken in getting these right.

7 Conclusions and Future Work

The CZT Java framework and XML format have been developed in order to improve tool support for the Z specification language, particularly for the ISO Standard for Z. It allows developers to easily develop new Z tools and to integrate existing tools via the XML interchange format. The CZT software is available from the Sourceforge web-site `http://czt.sourceforge.net` under an open source license.

Currently, we are developing a *rewrite rule* mechanism for user-defined AST transformations and for unfolding schema operators. We are also working on a central *section manager* subsystem, which will manage all the Z objects, the dependencies between them and the commands that transform them.

In the future, we would like people to develop many more translators from the CZT AST (standard Z) into older dialects of Z to give access to the existing Z provers, and into Alloy[21] for performing simple animations and counter-example generation.

We would also like to integrate the CZT tools with an integrated development environment, to provide a full WYSIWYG editing and analysis environment for

[21] See `http://alloy.mit.edu`.

Z. We have developed an experimental plug-in for JEdit[22] that does this, but a better long-term alternative may be the Eclipse environment,[23] or the ZEUS[24] system, which is an extension of Adobe Framemaker.

Acknowledgements

Thanks to Andrew Martin for proposing the CZT project, and to the dozen or more people who have contributed expertise and code over the last two years[25], particularly Jin Song Dong and his students at National University of Singapore, who have been working on the TCOZ extensions. Thanks to Ian Toyn for his XML DTD for Z, and for answering numerous questions about the ISO Standard for Z. Especial thanks to Tim Miller, who has implemented most of the Z and Object Z parser and typechecker.

References

1. ISO/IEC 13568: Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics. First edn. ISO/IEC (2002)
2. Utting, M., Toyn, I., Sun, J., Martin, A., Dong, J.S., Daley, N., Currie, D.: ZML: XML support for standard Z. In: ZB 2003: Formal Specification and Development in Z and B: Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003. Proceedings, Springer-Verlag Heidelberg (2003) 437–456
3. Smith, G.: The Object-Z Specification Language. Advances in Formal Methods. Kluwer Academic Publishers (2000)
4. Mahony, B., Dong, J.S.: Timed communicating Object Z. IEEE Transactions on Software Engineering **26** (2000) 150–177
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, USA (1995)
6. Mai, Y., de Champlain, M.: A pattern language to visitors. In: The 8th Annual Conference of Pattern Languages of Programs (PLoP 2001), Monticello, Illinois, USA. (2001)
7. Martin, A.C.: Acyclic visitor. In Martin, R.C., Riehle, D., Buschmann, F., eds.: Pattern Languages of Program Design 3, Addison-Wesley Longman Publishing Co., Inc. (1997)
8. Nordberg III, M.E.: Default and extrinsic visitor. In Martin, R.C., Riehle, D., Buschmann, F., eds.: Pattern Languages of Program Design 3, Addison-Wesley Longman Publishing Co., Inc. (1997)
9. Toyn, I., Stepney, S.: Characters + mark-up = Z lexis. In: ZB 2002: Formal Specification and Development in Z and B: 2nd International Conference of B and Z Users, Grenoble, France, January 23-25, 2002. Proceedings. Volume 2272 of LNCS., Springer-Verlag Heidelberg (2002) 100–119

[22] See `http://www.jedit.org`.
[23] See `http://www.eclipse.org`.
[24] See `http://www.cs.virginia.edu/~zed/zeus`.
[25] See `http://czt.sourceforge.net/people.html`.

10. Toyn, I., Valentine, S.H., Stepney, S., King, S.: Typechecking Z. In Bowen, J.P., Dunne, S., Galloway, A., King, S., eds.: ZB 2000: First International Conference of B and Z Users, York, UK, August 2000. Volume 1878 of LNCS., Springer (2000) 264–285
11. Spivey, J.M.: The Z Notation: A Reference Manual. Second edn. International Series in Computer Science. Prentice-Hall International (UK) Ltd (1992)
12. Legeard, B., Peureux, F., Utting, M.: Automated boundary testing from Z and B. In Eriksson, L.H., Lindsay, P., eds.: Formal Methods Europe, FME 2002. Volume 2391 of LNCS., Springer-Verlag (2002) 21–40
13. Toyn, I.: Information technology – Z formal specification notation – Syntax, type system and semantics. DRAFT TECHNICAL CORRIGENDUM 1, Corrections to use of Unicode. Available from `http://www-users.cs.york.ac.uk/~ian/zstan/IS.html` (2004) This draft has yet to be submitted for official ballot.