CZT Support for Z Extensions

Tim Miller[1], Leo Freitas[2], Petra Malik[3], and Mark Utting[3]

[1] University of Liverpool, UK
tim@csc.liv.ac.uk
[2] University of York, UK
leo@cs.york.ac.uk
[3] University of Waikato, New Zealand
{petra, marku}@cs.waikato.ac.nz

Abstract. Community Z Tools (CZT) is an integrated framework for the Z formal specification language. In this paper, we show how it is also designed to support extensions of Z, in a way that minimises the work required to build a new Z extension. The goals of the framework are to maximise extensibility and reuse, and minimise code duplication and maintenance effort. To achieve these goals, CZT uses a variety of different reuse mechanisms, including generation of Java code from a hierarchy of XML schemas, XML templates for shared code, and several design patterns for maximising reuse of Java code. The CZT framework is being used to implement several integrated formal methods, which add object-orientation, real-time features and process algebra extensions to Z. The effort required to implement such extensions of Z has been dramatically reduced by using the CZT framework.

Keywords: Standard Z, Object-Z, TCOZ, *Circus*, parsing, typechecking, animation, design patterns, framework, AST.

1 Introduction

The Z language [1] is a formal specification notation that can be used to precisely specify the behaviour of systems, and analyse them via proof, animation, test generation, and so on. Z was approved as an ISO standard in 2002, but currently there are few tools that conform to the standard.[1] The Community Z Tools (CZT) project [2] is an open-source Java framework for building formal methods tools for standard Z and Z extensions.

CZT[2] provides the basic tools expected in a Z environment, such as conversion between LaTeX, Unicode and XML formats for Z, and parsing, unparsing, typechecking and animation tools, with a WYSIWYG Z editing environment integrated within the jEdit[3] editor. There are also several more experimental

[1] CADiZ (http://www-users.cs.york.ac.uk/~ian/cadiz) is the only Z tool that conforms closely to the Z standard. It is freely available, but is not open-source and does not aim at supporting Z extensions.
[2] See http://czt.sourceforge.net.
[3] See http://www.jedit.org.

J. Romijn, G. Smith, and J. van de Pol (Eds.): IFM 2005, LNCS 3771, pp. 227–245, 2005.

tools under development, such as a Z-to-B translator and a semi-automated GUI-builder for Z specifications. However, the main design goal of CZT is to provide a framework which makes it easy to develop new Z tools. This paper describes how the framework also makes it easy to develop tools for extensions of Z.

In recent years, there has been an increasing interest in combining different programming paradigms within a uniform formal notation, where Z plays a central role. This has given rise to many Z extensions, which add features such as process algebras [3, 4, 5], object orientation [6, 7], time [8, 9], mobility [10], and so forth.

Among these extensions, CZT supports Object-Z [6], a specification language that extends Z with modularity and reuse constructs that resemble the object-oriented programming paradigm. Such constructs include classes, inheritance, and polymorphism. CZT supports Object-Z in the form of parsing, typechecking, and other facilities. CZT is also being used to develop extensions for Timed Communicating Object-Z (TCOZ) [8], which is a blend of Object-Z and Timed-CSP [11], as well as extensions for *Circus* [5], a unified refinement language that combines Z, CSP [12], and the refinement calculus [13], with Hoare and He's *Unifying Theories of Programming* (UTP) as the semantic background [14][4].

This paper describes the engineering techniques used in the CZT framework to maximise extensibility and reuse. Most of these techniques could also be applied to frameworks for other integrated formal methods, especially when the framework must support several different extensions of a common base language (like the role of Z in CZT).

In Section 2, we present a method for specifying an XML interchange format that maximises extensibility. Section 3 describes the automatic generation and design of the *Annotated Syntax Tree* (AST) classes. Section 4 presents a method for generating parsers, scanners, and other related tools for the different Z extensions, and Section 5 presents the design of the CZT typecheckers, which are tailored for extendibility and reuse. Section 6 briefly presents the CZT animator, ZLive, and discusses the possibility of using this to animate extensions to Z. Section 7 presents the design of the *specification manager*, an integral component of CZT that caches information about specifications to improve the efficiency of the tools. Section 8 gives an overview of related work. Finally, Section 9 concludes the paper and discusses the future of the CZT project.

2 XML Schemas

The first step in designing the CZT tools and libraries was the development of an XML schema that describes an XML markup for Z specifications (ZML) [15]. This is an interchange format that can be used to exchange parsed Z specifications between sessions and tools written in different languages.

Standard Z allows specifications to be exchanged using Unicode, LaTeX or email markup. However, implementing a parser for such specifications is a non-trivial task that can take several months. ZML, in contrast, can be parsed im-

[4] See `http://www.cs.york.ac.uk/circus/`

mediately since virtually all programming languages provide XML reading and writing libraries.

The idea of using XML for Z has also been explored in the Z/EVES theorem prover [16]. It allows one to create a customised theorem prover with additional tactics tailored for a particular specification by modifying the XML representation of the Z specification in Z/EVES [17]. The main problem however, is the lack of a common standard.

The XML schema for ZML was carefully designed, via consensus between several groups of interested people, by selecting the best features of the abstract syntaxes of CADiZ, Zeta and the Z standard. ZML supports several kinds of extensibility:

Extensible Annotations: Each Z construct can be *annotated* with arbitrary information, such as type information, comments, anticipated usage, and source-file location.

Extensible ASTs: This allows Z extensions to add new kinds of expressions, predicates, paragraphs, *etc.*

Extensible Schemas: The standard XML schema features, such as namespaces and importing, mean that Z extensions can be defined without modifying the original ZML schema.

The following strategies have been used to achieve these kinds of extensibility.

The *"any"* element can be used in an XML schema to enable instance XML documents to contain additional elements not specified by the schema. This concept has been used to define annotations. That is, an annotation to a term can either be one of the annotations defined in the XML schema for Z, or any other kind of data. New kinds of annotations can be added without changing the ZML schema. This allows a tool builder to decide what data makes sense for a particular tool. Tools that do not use a particular kind of annotation simply ignore those annotations.

A typical style of defining XML schemas or DTDs is to explicitly list the possible alternatives for expressions, predicates, *etc.* This makes it difficult to extend the syntax of ASTs to allow new kinds of expressions or predicates. In contrast, ZML uses *inheritance* (*substitution groups* in XML schema terminology) extensively throughout the XML schema. Abstract elements are used to provide placeholders for their derived elements. For example, the abstract element `Para` is the parent of all concrete Z paragraphs, such as axiomatic paragraphs (element `AxPara`), and free types paragraph (element `FreePara`). Other elements that contain paragraphs, like Z section (element `ZSect`), are defined to contain a *reference* to the abstract `Para` element. This allows any subtype of `Para` to be used instead. This has the same extensibility advantages as subtyping in object-oriented languages.

A Z extension can add new kinds of paragraphs, expressions and predicates, simply by extending these ZML inheritance hierarchies. It is important to note that this can be done without modifying the ZML schema file. Instead, the Z extension creates a new XML schema which *imports* the original ZML schema file, then defines the additional constructs using a new namespace. This means

that several separate extensions of Z can easily coexist. For example, the XML schema for Object-Z imports the ZML schema file, and defines a new paragraph for classes (element `ClassPara`) that is derived from element `Para` defined in the ZML schema. Instance documents of the Object-Z schema can now contain class paragraphs in addition to the standard Z paragraphs wherever an element `Para` is expected. Thus, an Object-Z specification in XML format can contain a mixture of Z and Object-Z constructs, such as:

```
<Z:ZSect>
  <OZ:ClassPara> ...  <Z:True/> ... </OZ:ClassPara>
</Z:ZSect>
```

This process of extending the XML schema can be done multiple times, so that even a Z extension can be extended. For example, the additional elements provided by the Object-Z XML schema are further extended by the TCOZ XML schema. Again, the definitions of the elements for TCOZ are encapsulated into a TCOZ XML schema file, and the ZML and Object-Z XML schemas do not need to be modified. Similarly, the *Circus* extension for CZT is encapsulated into a *Circus* XML schema file that extends the main standard Z schema. This approach of extension via inclusion is explored throughout the different layers of CZT tools. The resulting net effect is that once one package is finished, it can be directly extended through inheritance, hence simplifying the task of extending standard Z to a great extent.

The use of XML in CZT has proved to be an efficient and extensible solution for representing a Z specification and its extensions. The XML approach helps to clarify design decisions in a straightforward fashion. This representation is the key for the integrated development and exchange of information among different Z tools.

3 Java AST Classes

To manipulate Z *Annotated Syntax Trees* (AST) within Java (or any other programming language), we must convert ZML files into Java objects. This could easily be done using one of the Java XML reader/writer libraries, such as DOM, but this would result in a very generic interface to the Java objects — to the programmer they would appear to be an N-ary tree of `Element` and `Text` objects. This does not accurately reflect Z syntax or semantics, is not elegant, and is error-prone to use.

Instead, we provide a customised Java interface for each Z construct, with appropriately named *get* and *set* methods for each subcomponent. This makes programs more readable, and provides much stronger typechecking. However, there are some situations where the generic N-ary tree view is more convenient (for example, writing a deep copy procedure), so our Java interfaces also provide a low-level generic view of each node, via the following two methods:

```
Object[] getChildren();  // return all children of this node
Term create(Object[] args); // create a new version of this node,
                            // with the given children.
```

Having these two alternative views of each node of the AST gives the best of both worlds — one can write generic tree traversal algorithms using the above two methods, as well as readable and type-safe Z-specific syntax manipulations using the node-specific get and set methods.

In fact, these CZT Java AST interfaces and their implementation classes are automatically generated from the XML schemas described in the previous section using our code generator *GnAST* (GeNerator for AST). The generated code looks similar to the code produced by Java data binding tools like JAXB[5] or Castor[6]. While the main purpose of a Java binding tool is to provide the ability to convert from XML format to Java objects and vice versa, the main purpose of GnAST is to generate well-designed AST classes. For example, the AST classes generated by GnAST support an extensible variant of the visitor design pattern [18, 19].

The automatic AST generation from the XML schemas dramatically reduces the time required to develop a new Z extension, ensures a common style of interface, and improves maintainability. For instance, the complete AST folder representing standard Z contains around 420 Java files. GnAST has also been used to generate AST interfaces and classes for Object-Z, TCOZ, and *Circus*. In total, from the four XML schema files for standard Z and its extensions, GnAST automatically generates around 2300 Java files. This provides a very convenient and consistent way to obtain AST interfaces and classes for Z extensions that fit well into the AST for standard Z.

The visitor design pattern [18, 19] makes it very easy to write tools like typecheckers and printers, which need to traverse an AST. It allows new traversal operations to be defined without modifying the AST classes. To define a new operation, all one needs to do is to implement a new visitor class.

The visitor design pattern used in CZT has been described in detail in [2]. It is a variant of the *acyclic visitor* [20] pattern and the *default visitor* [21] pattern. Its additional advantages over the standard visitor pattern are that it allows the AST interfaces and classes to be extended without affecting existing visitors, and that it allows a visitor to take advantage of the AST inheritance relationships. For example, a copy visitor that copies an AST can provide a default behaviour for `Term`, the base of the AST inheritance hierarchy. Since AST classes for extensions also derive from `Term`, this copy visitor works for any extension. On the other hand, if the default copy behaviour is not wanted for a particular extension class, say `XYZ`, one can simply add a `visitXYZ` method to the copy visitor, and that method will be used instead of the default `visitTerm` method.

This has a big impact on the applicability of visitors for extensions like Object-Z, TCOZ, and *Circus*. Firstly, it ensures that the Z AST classes can be extended without having to modify existing Z visitors like typechecker, printer, *etc.* Secondly, it makes it easy to extend existing visitors to handle Z extensions — one can simply define a new visitor class which inherits behaviour from an

[5] See `http://java.sun.com/xml/jaxb/`
[6] See `http://www.castor.org/`

existing Z visitor and adds a few methods for the new or changed language constructs. Finally, by defining default behaviours for abstract classes such as `Expr` or `Decl`, it is possible to implement tools that are applicable to all Z extensions.

In conclusion, the CZT AST classes provide:

A Choice of Coding Style: One for **generic low-level** algorithms and the other for **node-specific high-level** algorithms.

Automation: the AST classes are generated automatically by **GnAST**.

Reuse of Algorithms: The CZT **visitor pattern** allows AST traversal algorithms to be reused and extended in flexible ways.

Extensibility: the standard Z AST can easily be extended by defining new **XML schemas**.

4 Parsers, Scanners, and Related Tools

CZT includes a suite of important tools for operations such as parsing, typechecking, and markup conversion. In addition to a parser and typechecker for Z, an Object-Z parser is provided, and *Circus* and TCOZ parsers, as well as an Object-Z typechecker, are under development. The Object-Z, TCOZ, and *Circus* tools extend the Z tools by adding support for the additional constructs these languages provide. As each language is an extension of Z, it is tempting to just keep adding to the tools for each extension, and use the largest superset of all extensions. For example, use the TCOZ tools to parse and typecheck Z. However, this has two distinct problems. Firstly, one aim of the CZT project is to create tools that strongly conform to the Z standard. However, allowing extra constructs to be parsed and using different type-rules will break the strong conformance. Secondly, the extensions of Z are not linear. For example, Object-Z extends Z with class paragraphs, and TCOZ extends Object-Z with concurrency operators, but *Circus* extends neither of these — only Z. Therefore, CZT requires an approach that produces separate tools for each Z extension, maximises the commonality between the parsers, and minimises versioning and maintenance problems via reuse.

4.1 Parsers and Scanners

CZT includes parsers for standard Z specifications given either in Unicode or LaTeX markup. Support for email markup is planned. Java Cup[7] is used to generate the CZT parsers from an LALR grammar, and JFlex[8] is used to generate the scanners.

Unfortunately, it is quite difficult to reuse code from an automatically generated scanner or parser, and neither Java Cup nor JFlex explicitly supports inheritance for parser or scanners respectively. To avoid duplicated code, XML templates that contain the different parser and scanner variants are used. From this, the different source files for each Z extension are generated using XSLT[9],

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

[8] See `http://jflex.de/`

[9] See `http://www.w3.org/TR/xslt`

a language for transforming XML documents. This maximises the commonality between the parsers and minimises versioning and maintenance problems.

All parser and scanner variants are maintained in master XML files. Each master file contains several XML tags that are used for substituting text for each Z extension. For example, the `<package/>` tag is placed wherever one would normally write the Java package name, so that each parser and scanner can be contained in their own package. The tags `<add:`*extension*`>` and `</add:`*extension*`>` are used to wrap around code that are specific to particular Z extensions. Thus, to add a new type of expression to the Object-Z parser, one would add a new production to the appropriate grammar rule in the master file, and place it between the `<add:oz>` and `</add:oz>` tags. In other programming languages, conditional compilation could be used to achieve the same result. However, as Java does not support conditional compilation, we use the XML template translation approach.

To generate the individual Java Cup files for each extension of Z, XSLT is used to include the necessary code, and to substitute in values for tags. For example, to generate the Object-Z parser, XSLT is applied to the master file, and supplied with the three arguments below:

1. `"class"` substituted with `"Parser"`.
2. `"package"` substituted with `"net.sourceforge.czt.parser.oz"`.
3. code in `"oz"` tags to be included.

Similar rules are specified for each parser and scanner variants. The result is a series of Java Cup and JFlex files, one for each language, which can then be used to generate the parser and scanner code.

The use of XML templates enables parsing code to be reused and easily maintained. Extending the parser and scanner for a new language can be done by just adding the respective grammar and lexer rules together with few modifications such as those parameters above. For example, we are experimenting the incorporation of the available *Circus* parser [22] rules within the flexible CZT framework. The obvious advantages are the widely tested and supported standard Z classes, LaTeX markup and Unicode, visiting and other facilities.

4.2 Multiple Markups

CZT supports multiple markups for each Z extension. The different markup languages suit different communities. For example, LaTeX is preferred by researchers, while Unicode WYSIWYG editing might be more attractive for students or industrial users. At present, Unicode, LaTeX, and the XML format are supported. Adding additional markups is straightforward, as this section will present. XML markup is not considered any further because it can be parsed immediately using existing XML parsers. CZT uses JAXB[10] to unmarshal an XML document into a tree of Java objects, and then uses the visitor design pattern to convert this tree into an AST.

[10] See `http://java.sun.com/xml/jaxb/`

In order to avoid having to provide a parser for each markup language, all specifications are first translated into Unicode and subsequently parsed by a Unicode parser[11]. This also makes sure that names in the AST are markup independent: they are represented in Unicode independently on the actual markup used in the source document. This is a necessary precondition of allowing different sections of a specification to be written in different markups. If a parser for a new markup is required, only a translator to Unicode needs to be implemented.

A consequence of this architecture is that extensions of Z need to support at least Unicode. CZT provides a Z Unicode scanner, which performs lexical analysis on a Unicode stream and breaks it into the necessary tokens. A scanner for a Z extension can be derived by adding additional scanner rules to the CZT scanner template as described above. In order to support LaTeX markup, it is convenient to provide a LaTeX toolkit section for a given extension that defines new operators for that language. In addition to defining new operators, these LaTeX markup documents contain LaTeX markup directives [1, 2] used to specify how certain LaTeX commands are to be converted into Unicode. The LaTeX to Unicode translator parses these definitions and converts each LaTeX command into the corresponding Unicode sequence. However, LaTeX `\begin{xxx}` and `\end{xxx}` environments cannot be defined using LaTeX markup directives. If a Z extension needs to provide new LaTeX environments, the LaTeX to Unicode converter needs to be adapted directly. Again, this is possible by adding new rules to the converter template file.

An additional benefit of this approach is that it reduces the number of converters needed between languages. That is, CZT currently implements LaTeX to Unicode and Unicode to LaTeX converters. In the future, we plan to implement an email to Unicode converter to allow parsing of specifications written in email. Using this and the Unicode to LaTeX converter, we could convert email to LaTeX. So, using an intermediate format reduces the number of converter tools that need to be implemented from $M * (M - 1)$ to $2 * (M - 1)$, in which M is the number of markup languages supported.

In conclusion, CZT supports extensions to parser and scanners using:

XML Templates for Code Sharing: XML templates are used to maximise code reuse for the parser and scanner scripts.

Unicode as an Intermediate Format: Unicode is used as an intermediate format to simplify the process of writing scanners and reduce the number of converters needed between markups.

5 Typecheckers

Typecheckers in CZT are written in a different way from the parsers and scanners. Each Z extension has its own typechecker, and while reuse is of high importance, using XML templates is unnecessary because unlike the parsers, Java interfaces and inheritance can be used to extend the typecheckers.

[11] See [2] for a more detailed description of the parser architecture.

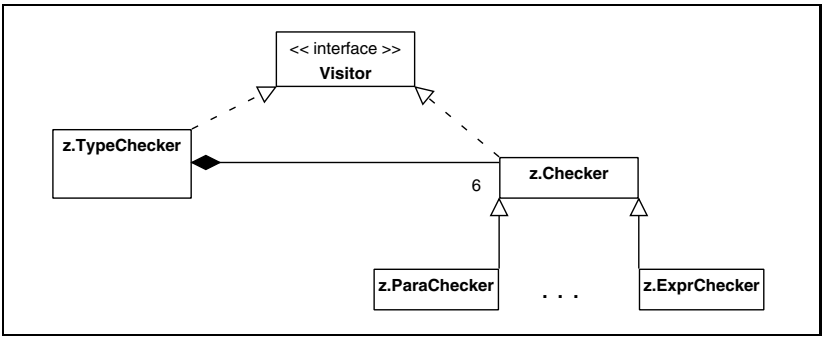

Fig. 1. UML class diagram for Z Typechecker

The Z typechecker is the base implementation. When a Z specification AST is passed to this typechecker, it applies all the typechecking rules and, if the specification is type-correct, it returns TRUE and annotates the original AST with type information as defined in the ISO standard [1–Section 10]. If the specification contains type errors, the result is FALSE, the AST is unchanged and a list of error messages describing the type errors (including their line and column position) is made available.

Most of the typechecker is written using visitors, which can be extended as discussed in Section 3. While it is tempting to write the typechecker as one large visitor, this would create maintenance problems as this visitor would be quite large and monolithic. So we use a more sophisticated and extensible design, shown in Fig. 1.

This breaks up the overall task of typechecking into several (currently six) smaller `Checker` visitors — each subclass of `Checker` typechecks a different kind of syntactic construct such as paragraphs, predicates, expressions, *etc.* The `Checker` class itself defines some shared resources, such as typing environments and the type unification facilities, as well as common "helper" methods used throughout the implementation such as error reporting. In addition, each checker subclass object has a reference back to the top-level `TypeChecker` object, which has links to all the checkers — this allows one checker to call another via the `TypeChecker` object.

For example, for typechecking a schema text of an `AxPara`, the `ParaChecker` class, which typechecks Z paragraphs, needs to typecheck both the declarations and the predicate parts of the schema text. Although visiting through the given AST is the general solution, the typechecking of the declarations part is delegated to the `DeclChecker` class, whereas the typechecking of the predicate part is delegated to the `PredChecker` class. The `DeclChecker` in turn uses the `ExprChecker` to ensure that expressions defining the declaring variables type are well-formed. Because each of these visitors share the same `TypeChecker` reference, and hence the same references to type environments, the declarations added to the type environment by the `DeclChecker` will be accessible by the other checkers.

There are a few additional classes that are used in the typechecker, but not shown in Fig. 1, such as the `UnificationEnv` class that performs the unification of two types for type inference and for checking type consistency.

The advantages of this typechecker design include:

- Methods that are common to all the `Checker` subclasses can be put in the `Checker` superclass. Data that is shared between the checkers can be managed by the `Typechecker` class and made accessible to the checkers in a controlled way via access methods.
- Splitting the overall typechecking task into several parts increases modularity and maintainability, and provides better encapsulation for the different checkers. This aids debugging and allows development of the checkers to be somewhat independent (for example, assigned to different teams or to different iterations of an agile lifecycle).
- Each `Checker` subclass is typechecking similar kinds of nodes (*e.g.*, all expressions), so can have a uniform visiting protocol, which increases regularity and helps to reduce errors. For example, all the visitor methods of the `ParaChecker` class, which typechecks Z paragraphs, return a `Signature` of the name and type pairs declared in that `Para`. In contrast, the `ExprChecker` class typechecks expression nodes and all its visitor methods return a `Type` with resolved reference parameters in which type unification has already been performed.
- By defining several `Checker` subclasses over the same kinds of AST nodes, it becomes easy to have multiple algorithms over the same syntax nodes. For example, post-checking for unresolved set and reference expressions, which may introduce an unresolved type, is implemented as a second kind of `ExprChecker`. This post-typechecking pass ensures that all implicit parameters such as generics actuals have been completely determined. This would not be possible with a single monolithic visitor design, because one could not have two `visitRefExpr` methods in the same visitor.

Fig. 2 shows how this design is extended to define a typechecker for a Z extension — Object-Z in this case. A new package (oz) is created for the Object-Z typechecker. In this package, a new `oz.Checker` class is implemented, which inherits the base `z.Checker` class. In this new class, any common methods that are to be used by the Object-Z typechecker are implemented, and existing methods are overridden or overloaded if additional functionality is needed. Then, new `Checker` subclasses are created, one for each kind of language entity that requires Object-Z-specific typechecking. Each of these checkers (the `oz.`*`XXXX`*`Checker` subclasses in Fig. 2) implement the visitor methods only for Object-Z constructs and for any Z constructs that require additional Object-Z-specific checking. The remaining standard Z constructs are handled by delegation to the original `z.`*`XXXX`*`Checker` object.

It is interesting to see how this delegation is achieved, given that Java does not support multiple inheritance. We rely on the general visiting protocol described in Section 3 and in [2]. For example, the `oz.ExprChecker` class catches all Object-Z-specific expressions. It also implements an additional `visitExpr`

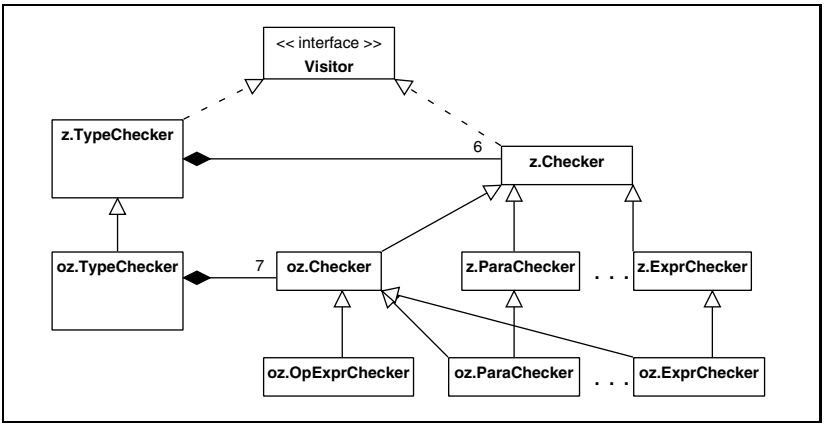

Fig. 2. UML class diagram for Object-Z Typechecker

method which "catches" all remaining `Expr` AST nodes and uses the visitor from `z.ExprChecker` to check those nodes.

```
private z.ExprChecker zExprChecker_;
...
public Object visitExpr(Expr expr) {
  return expr.accept(zExprChecker_);
}
```

The Z typechecker has a reference to a `z.ExprChecker` object, but in the Object-Z typechecker, this points to an `oz.ExprChecker` instead. When an Object-Z expression is typechecked, it is handled directly by the `oz.ExprChecker` instance. When a standard Z expression is typechecked, the above `visitExpr` method is called, delegating the typechecking to an instance of `z.ExprChecker`. Any subexpressions of the Z expression are passed back to the top-level typechecker, which uses the `oz.ExprChecker` instance, to ensure that Object-Z subexpressions are checked correctly.

This also allows type-rules to be overridden. For example, a selection expression, $a.b$, in standard Z requires that a is a schema binding, whereas in Object-Z, a can also be an object. The `ExprChecker` in the Object-Z implements the visit method for such expressions, and this method first checks if a is an object, and if not, delegates the call to the Z typechecker.

Although this is an unusual design, it has proven to provide good and elegant support for extension. An alternative approach that we considered was for the Object-Z checkers to directly subclass the Z checker subclasses (*e.g.*, `oz.ParaChecker` to inherit `z.ParaChecker`). However, this would have meant that the common code implemented in the current `oz.Checker` class would have had to have been implemented in the base `Checker` class, which would have resulted in an undesirable strong coupling between all of the typecheckers.

Other components are extended using inheritance. For example, the class `UnificationEnv`, which is responsible for type unification, is extended by overriding its `unify` method to handle the new Object-Z types, while using the superclass's `unify` method for standard Z types.

Our experience is that the above extensible typechecker design makes it much easier to build multi-lingual typecheckers. That is, a family of typechecker objects for Z and various extensions of Z. For example, a static checker for Circus that checks some context-sensitive rules such as variable and action declaration scope has been developed following the guidelines for Z and Object-Z typecheckers. This took only three to four days to develop and the task was made significantly easier because of the code reuse and elegant object-oriented design of the CZT typechecker. The information collected by this static checker is being used as an initial environment for the *Circus* operational semantics [23]. In the future, this static checker can also be used as the basis for a full *Circus* typechecker; the type-rules for *Circus* are under development in [24]. An obvious advantage of reusing the base Z typechecker is that the *Circus* typechecker will already enforce standard Z typechecking conformance. Therefore, one can concentrate on the implementation of new type-rules for *Circus* in this available skeleton for a *Circus* typechecker.

In conclusion, CZT supports extendibility in its typecheckers by:

Using Multiple Visitors: A separate visitor is used for each group of type-rules; this provides a straightforward way to implement type-rules for new constructs (by adding new visitors), or override existing type-rules (by subclassing existing visitors).

Sharing Common Code via Inheritance and Delegation: Methods used throughout the typechecker are shared in several abstract super classes that are reused via both inheritance and delegation.

Sharing Resources: The `TypeChecker` class is used by visitors to provide access to common resources and to other visitors.

6 Animation

Further to parsing and typechecking standard Z and its extensions, CZT also provides animation facilities with its ZLive tool. Z animation is particularly useful for testing, rapid prototyping, and experimenting with specifications. In addition, given suitable restrictions to finite state models, an animator can be used for finite theorem proving (or theorem testing), and model checking. An extensive discussion and comparison of Z animation tools available is given in [25].

6.1 Extending ZLive

ZLive is an animator capable of evaluating predicates and expressions using mode analysis [26]. Mode analysis consists of including additional (type and formulae ordering) information not present in specifications, which enable evaluation and

animation. The architecture of ZLive is an evolution of a previous Z animator implemented in Haskell[12].

The ZLive architecture is divided into six tasks. Firstly, a target expression is given. Secondly, the definitions are unfolded so that schema inclusions are grounded to base terms. Next, the unfolded definitions are flattened into a normal form of atomic predicates. After that, possible evaluation modes are calculated for each flatten predicate. These moded-predicates are then reordered according to the cheapest solution order in terms of number of solutions. Finally, all solutions are lazily enumerated as requested.

ZLive currently supports basic logic and arithmetic operators (*e.g.*, $\forall$, $\exists$, $\neg$, $\land$, $-$, $+$, $*$, $\leq$, $<$, `div`, `mod`, `succ`), set representations (comprehension, ranges, and displays), unfolding of simple definitions, tuples, and schema bindings. For efficient execution, the main issue is to find a good reordering of atomic predicates which minimises the expected enumeration time. Currently ZLive uses a naive algorithm for this, but in the future we expect to implement a best-first or A^* path-finding algorithm.

It is desirable to provide animation facilities for Z extensions as well as for standard Z. To extend ZLive to animate a new Z extension, there are three possible approaches:

Explicit Inclusion: Animation support for each new language construct, including any new evaluation algorithms, is directly added to ZLive by adding new Java classes and methods. This would use interfaces, inheritance and visitors to achieve an extensible architecture, similar to the CZT typechecker.

Transformation to Standard Z: If each new construct of the Z extension can be transformed back into standard Z using rewriting rules, then ZLive can be used directly on the result of that translation. This approach is being used to develop an Object-Z animator, with Object-Z objects being transformed into Z bindings, *etc.* This approach of rewriting specifications is similar to the Z refinement calculus [27, 13].

Meta-Level Animation: If the operational semantics of the new language can be given in standard Z, one can use ZLive directly to animate the new Z extension by animating its operational semantics. Although this is a meta-level approach to execution, which usually results in very slow performance, the performance impact should be less in this case, because any standard Z constructs within the Z extension can animated efficiently and directly by ZLive. That is, only the new constructs have to be animated by the slower, meta-level approach. This approach is taken for animating the operational semantics of *Circus* [23].

Depending on the new language constructs to be animated, these possibilities can be combined.

[12] See `http://www.cs.waikato.ac.nz/~marku/jaza/`

6.2 Extension Example: Animating *Circus*

We are currently experimenting using ZLive within the development of a model checker for *Circus* [28]. Among other aspects, we are particularly interested in integration of model checking and theorem proving facilities for *Circus*. In this direction, animation plays an important part in the evaluation of Z terms used to describe state aspects of dependable and distributed systems.

The *Circus* model checker architecture is divided into four main tasks as shown in Fig. 3. The first two involve parsing a *Circus* specification in LaTeX to produce an CZT AST, and typechecking to produce an annotated AST_+. They use the CZT parser and typechecker described in earlier sections. The last two stages involve compilation and refinement search. From the annotated AST_+ the compiler builds a *Predicate Transition System* (PTS) that finitely represent (possibly infinite specifications) base on the operational semantics of *Circus* [23]. Both the PTS and the AST_+ are given to the model checker engine that integrates refinement model checking algorithms [29, 30] together with theorem proving and debugging functionalities[13]. The result is a (possibly empty) set of witnesses representing failed refinement conditions. More details of this architecture can be found in [28].

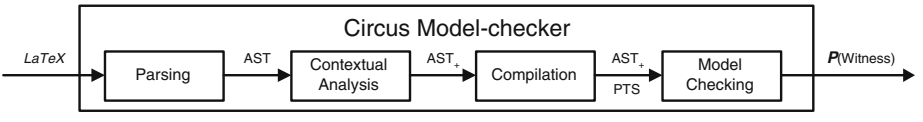

Fig. 3. *Circus* Model Checking Stages

In this architecture, ZLive is used from two different perspectives: (i) to animate the Z part of *Circus* specifications, and (ii) to evaluate the operational semantics of *Circus* given in Z, while performing the model checking search.

To implement the first perspective, we are extending ZLive via *direct inclusion* of several Z constructs (like θ and some schema operators) that are frequently used in *Circus* specifications but not yet implemented by ZLive. To implement the second perspective, we are using the *meta-level animation* approach to animate the operational semantics of the CSP parts of *Circus*.

The *transformation to standard Z* approach could also be used to animate the CSP parts of *Circus*. To have confidence in the correctness of this approach, it would be desirable to have correctness proofs for the rewriting laws. As *Circus* is heavily based on the notion of stepwise refinement, this transformation approach would fit nicely with the philosophy of *Circus*. Work in this direction of a refinement calculus for *Circus* is under development [31]. It also includes the basis for a *Circus* theorem prover [32].

The theorem proving module in the *Circus* model checker (which is used both in the compiler and refinement engine), dispatches requests for evaluation

[13] See `http://www.cs.york.ac.uk/circus/model-checker`

of Z expressions and predicates. These are either verification conditions over the state operations defined in Z, or possible enabling paths available for investigation from the behavioural actions given using CSP. They are both given as standard Z statements from the operational semantics of *Circus*. At this point, theorem proving is usually necessary to discharge proof obligations, and transform expressions or predicates. Nevertheless, for specifications with simple state operations, animation is also a good idea that could improve the automation levels of the model checking process.

The role ZLive plays in this scenario is to tackle the requests to evaluate Z expressions and predicates from the theorem proving module within the compiler and refinement engine. As the operational semantics of *Circus* is given in Z itself, we can use ZLive as a meta-level animator for simple specifications, hence enabling automatic model checking of state-rich *Circus* specifications.

With a few improvements and extensions to the current implementation of the schema calculus in ZLive, it should be possible to automatically model check simple-state *Circus* specifications within ZLive. Furthermore, as the theorem proving integration architecture of the *Circus* model checker allows pluggable solutions suitable for individual contexts, if ZLive cannot handle some complex *Circus* specifications, we can still resort to some alternative solution such as SAT solvers, and general-purpose theorem provers.

These *Circus* tools, some of which are currently under development, give some good examples of how to integrate different CZT tools across different notations and tool boundaries, from standard Z parsing through to extended typechecking and animation for *Circus*.

7 Specification Manager

One of the core components of the CZT framework is the *specification manager*, which is an extensible repository for formal methods objects. Most of the tools mentioned in the previous sections use the specification manager to enquire about specific aspects of a specification. For example, to be able to parse a Z section, the Z parser needs the operator definitions of the parent sections. In order to typecheck a Z section, the section must be parsed and the parents of that section typechecked. To print a Z section in LaTeX markup, the operator definitions and LaTeX markup directives of the parent sections are needed.

While it would be possible to hard-code these dependencies and let, for example, the LaTeX markup printer call the parser for the parent sections directly, it is more convenient, extensible, and efficient to have a central repository that is responsible for this task. The CZT specification manager caches information about all the specifications and Z sections that are being processed and automatically runs tools such as markup converters, parsers and typecheckers when necessary. The caching of the parsed form of commonly used objects, such as standard toolkit sections, avoids repeated parsing and analysis of these objects and can give significant performance improvements.

Abstractly, the cache is a mapping from a key to the actual data. The key is a `(String, Class)` pair, where the `String` is usually the name of the section, and the `Class` is the Java class of the type of data associated with this key. This allows several different kinds of objects to be associated with one section, and provides some dynamic type security. For example, the Z parser adds the AST of a specification it has parsed to the specification manager. The type of a Z section in Java is `ZSect.class`. Thus the AST for a section called `foo` is cached under the key `(''foo'', ZSect.class)`.

The CZT specification manager supports two important kinds of extensibility:

Type Extensibility: Z extensions can easily use the specification manager to store new types of information, since the flexible `(String, Class)` key system allows arbitrary Java objects to be stored and retrieved. That is, the kinds of objects managed by the specification manager are open-ended, rather than being a fixed set of Z-related objects.

Command Extensibility: A Z extension can easily add or override the *default commands* of the specification manager. The default commands of the specification manager are responsible for automatically computing requested objects; they are implemented using the command design pattern [18]. For example, if the AST for section `foo` (*i.e.*, data of type `ZSect.class`) is required and has not already been cached, the Z parser is called by the specification manager in order to parse the specification file containing section `foo`. Here, the Z parser is the default command to compute data of type `ZSect.class`. A Z extension that needs to use a different parser can simply override the default command associated with the type `ZSect.class`. For example, the specification manager can be configured to always use the Object-Z parser.

A major advantage of this default command approach is that it simplifies tool development and makes tools more flexible, because a particular tool does not have to know which other tools to use in order to find information about a section — it simply requests the key that it wants and the specification manager will locate the information if it is able. This gives a more flexible, *plugin* style of tool development.

8 Related Work

Integrated formal methods frameworks have been investigated in the past. Anderson *et al.* [33] discuss a framework for integrating different formal methods tools. However, their aim is to specify generic interfaces to support integration of formal methods tools. Three types of interfaces are used: between the engineer and the tools; between cooperating tools; and between the tools and the project environment. They achieve this by using an *Encapsulation Toolkit* to allow a formal methods tool to communicate with other components in an intermediate format, and an *Active Document Toolkit* to allow communication between tools and their human users. The goals of this project are different to CZT, which

aims to provide components for Z tools that can be extended and integrated into the project or other tools.

Brillant[14] is an open-source project with similar aims to CZT, but for the B method. It aims to integrate several existing projects (BCaml, jBTools and ABTools), which all contain parsers and typecheckers for various dialects of B. Brillant is an approach to integrating these tools in a loosely-coupled style, with tools being written in several different languages (OCaml, Java and XSLT) and communicating via a common XML format for B machines. Brillant includes a translator from UML to B, plus some experimental B extensions (Event B and a real-time extension of B based on the duration calculus), but the extensions seem to be designed on an individual basis, rather than being tightly integrated extensions of a core architecture like in CZT. The extensible architecture of CZT, and of course, the consistent use of Java for writing the tools, enables a higher degree of reuse.

Other formal methods toolkits exist, such as the RODIN project[15] for the B specification language, and the Overture toolset[16] for VDM, but they focus on providing specific tool support for their respective languages, whereas CZT aims to provide extensible components that can also be used by other tools.

Projects such as Eclipse[17] and UQ* [18] are projects aimed at providing generic language-based environments for software development. However, these projects are not tailored towards formal methods, and provide support for generic languages, leaving the development of parsers, typecheckers, and other language-specific tools up to users who want such support. CZT is exactly the opposite of this, in that it focuses only Z and various Z extensions, allowing specific components, such as parsers and typecheckers, to be included within the framework. Therefore, CZT could be integrated into the Eclipse or UQ* environments.

9 Conclusions and Future Work

In this paper, we have presented a variety of reuse and extensibility mechanisms that makes the CZT framework an ideal starting point to develop new integrated formal methods tools for Z and its extensions. We have shown how the XML schemas for Z, and for extensions of Z, support reuse and extension of the Z language. They also enable automatic generation of Java AST classes with two levels of interface, and a consistent implementation of the CZT visitor pattern.

Using examples from Object-Z, TCOZ, and *Circus*, we have discussed several practical strategies and techniques that allow the CZT tools like parsers, typecheckers, and animators developed for standard Z to be reused within these Z extensions in a way that minimises code duplication and maintenance. The

[14] See `https://gna.org/projects/brillant`.
[15] See `http://rodin-b-sharp.sourceforge.net/`.
[16] See `http://www.overturetool.org/`.
[17] See `http://www.eclipse.org/`.
[18] See `http://www.itee.uq.edu.au/~uqstar/`.

strategies and techniques presented can also help developers of integrated formal methods tools not based on Z to make their framework as extensible as possible.

We plan to develop additional tools for Z and its extensions, as well as extending the CZT framework itself. For instance, extensions of ZLive providing Object-Z constructs, schema unfolding, predicate reordering, rewriting rules, and a tactic language in the spirit of ANGEL [34] are on our agenda. These improvements would enable a basis for an extensible theorem prover for standard Z and its extensions that is open-source and cross-platform.

Z parsing and typechecking is neither a novel idea, nor a unavailable resource. Nevertheless, flexible and integrated open-source support for ISO standard Z heavily focused on strong conformance and extensibility has not previously been available. The **philosophy CZT advocates is simple: provide an open-source framework with a set of tools for editing, parsing, typechecking and animating formal specifications written in Z, with support for Z extensions**. As new extensions are included and the framework matures, we expect it to become the common base for a growing number of strongly conforming tools for Z and its extensions.

References

1. ISO/IEC 13568: Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics. First edn. ISO/IEC (2002)
2. Malik, P., Utting, M.: CZT: A Framework for Z Tools. In Treharne, H., King, S., Henson, M., Schneider, S., eds.: ZB 2005: Formal Specification and Development in Z and B: 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005. Proceedings, Springer-Verlag (2005)
3. Fischer, C.: How to combine Z with process algebras. Technical report, University of Oldenburg (1998)
4. Fischer, C.: Combination and Implementation of Process and Data: from CSP-OZ to Java. PhD thesis, University of Oldenburg (2000)
5. Woodcock, J., Cavalcanti, A.: A Concurrent Language for Refinement. 5th Irish Workshop on Formal Methods (2001)
6. Smith, G.: The Object-Z Specification Language. Advances in Formal Methods. Kluwer Academic Publishers (2000)
7. Cavalcanti, A.L.C., Sampaio, A., Woodcock, J.C.P.: Unifying Classes and Processes. Journal on Software and Systems Modelling (2005) To appear.
8. Mahony, B., Dong, J.S.: Timed Communicating Object-Z. IEEE Transactions on Software Engineering **26** (2000) 150–177
9. Sherif, A., He, J.: Toward a Time Model for *Circus*. In George, C., Miao, H., eds.: ICFEM 2002. Volume 2495 of LNCS., Springer-Verlag (2002) 613–624
10. Tang, X., Woodcock, J.: Towards Mobile Processes in Unifying Theories. In Cuellar, J., Liu, Z., eds.: SEFM2004: the 2nd IEEE International Conference on Software Engineering and Formal Methods. (2004) 44–53
11. Schneider, S., Davies, J.: A Brief History of Timed CSP. Theoretical Computer Science **138** (1995) 243–271
12. Roscoe, A.W.: The Theory and Practice of Concurrency. 1st edn. International Series in Computer Science. Prentice-Hall (1997)
13. Morgan, C.: Programming from Specifications. Second edn. Prentice-Hall (1994)

14. Hoare, C., Jifeng, H.: Unifying Theories of Programming. First edn. International Series in Computer Science. Prentice-Hall (1998)
15. 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 (2003) 437–456
16. Meisels, I., Saaltink, M.: Z/Eves 1.5 Reference Manual. ORA Canada. (1997) TR-97-5493-03d.
17. Saaltink, M., Meisels, I.: The Core Z/Eves API (DRAFT). Technical Report TR-99-5540-xx, ORA Canada (2003)
18. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, USA (1995)
19. 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)
20. 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)
21. 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)
22. Barbosa, A.: A Parser for *Circus*. Graduation Research Project (2002)
23. Woodcock, J., Cavalcanti, A., Freitas, L.: *Circus* Operational Semantics. In: Proceedings of Formal Methods Europe. (2005)
24. Xavier, M.: *Circus* Type-checker. Master's thesis, Universidade Federal de Pernambuco, Brazil (2006) In preparation.
25. Utting, M.: Data Structures for Z Testing Tools. Technical report, University of Waikato, Hamilton, New Zeland (1999)
26. Winikoff, M.: Analysing modes and subtypes in Z specifications. Technical Report 98/2, University of Melbourne, Department of Computer Science, Parkville, Victoria 3052, Australia (1998)
27. Cavalcanti, A.: A Refinement Calculus for Z. PhD thesis, Oxford University (1997) Also published as a PRG Technical Monograph at `web.comlab.ox.ac.uk/oucl/publications/monos/prg-123.html`.
28. Freitas, L.: Model Checking *Circus*. PhD thesis, Univeristy of York (2005) To appear in October 2005.
29. A. W. Roscoe: Model checking CSP. In book: A Classical Mind: Essays in Honour of C. A. R. Hoare (1994) 353–378
30. Cleaveland, R., Hennessy, M.: Testing Equivalence as a Bisimulation Equivalence. Formal Aspects of Computing **5** (1993) 1–20
31. Oliveira, M.: A Refinement Calculus for *Circus*. PhD thesis, University of York (2005) To appear in December 2005.
32. Oliveira, M., Cavalcanti, A., Woodcock, J.: Unifying Theories in ProofPowerZ. draft, Univeristy of York (2005)
33. Anderson, P., Goldsmith, M., Scattergood, B., Teitelbaum, T.: An Environment for Integrating Formal Methods Tools. In: User Interfaces for Theorem Provers. (1997)
34. Martin, A.P., Gardiner, P.H.B., Woodcock, J.C.P.: A Tactic Calculus. Formal Aspects of Computing **8** (1996) 244–285