Towards
Requirements for Community Z Tools

Andrew Martin
(incorporating Mark Utting's suggestions)

December 2001

This document attempts to collect together some of my ideas and those of others expressed in the `czt-project` mailing list recently. It is very definitely a draft document: comments will be welcomed, and changes may be anticipated. If it sometimes seems to cross the line from requirements specification to design, that is a common failing among users who think they know what they want…

Ideally this paper should be published at a conference or in a journal, in the names of those who contribute significantly to its development. In the fullness of time, it may split into a number of separate documents. Many of Mark Utting's ideas have been incorporated; some seemlessly, some so far not (and identified as such). His thoughts on design and architecture are perhaps implicitly present, but not reproduced.

I began by calling this the Community Z Tools Project; I think perhaps *Community Z Tools Initiative* might be a more appropriate term. Within the following description, a number of distinct projects can be identified — descriptions of such projects would be welcomed.

The requirements have been grouped into a number of 'Levels', with Level 1 as a collection of pieces of infrastructure, and Level 2 the beginnings of an integrated GUI. Level 0 entails the design and identification of file formats and programming interfaces. This does not necessarily imply a linear progression through the levels with time. Experience in Level 1 may feed back into Level 0, and so on.

1 Context

1.1 State of the art

Many have developed Z tools over the last fifteen years. A few have been commercial ventures; others have been development or research projects of varying levels of sophistication. Few of the commercial ventures have proved financially viable; few of the research projects have either received

sustained development or been archived in such a way as to be accessible to other researchers or users.

The Z notation continues to attract considerable interest from educators and researchers, as well as industrial application in a number of areas. Production of simple tools may not be commercially attractive; it is hardly a research exercise, either: all the 'proof of concept' work has been done. Nevertheless, such tools have great potential to aid learners in making sense of Z, and if well-designed could provide a useful platform for advanced and speculative research.

Z specifications are richly visual; the schema boxes make a Z document distinctive. The mathematics used in Z is expressed concisely (some would say tersely), in the best traditions of mathematical argument, using a large collection of symbols. Whilst these symbols may be the bane of a student's life, they contribute much to Z's ability to make complex descriptions in a few lines, and for users to examine and reason with small and comprehensible terms. Modern desktop computers can readily display these notations, yet many formal methods tools remain stubbornly text-based, at best complicating the learning process and at worst completely obscuring the possible clarity of expression.

1.2 Standard Z

Considerable effort over the last decade has contributed to the development of an ISO standard for the Z notation. In most regards, the *de facto* standard of Spivey's *Z reference manual* is a subset of Standard Z. Other writers have extended Z in various ways, typically incorporating explicit object-orientation or process-algebraic ideas.

Any new tools produced under this initiative should seek to conform to Standard Z. Designs should have flexibility, however, to support other dialects and experimental extensions to the language. In its long development, Z has never been a 'closed' fixed language, but rather a framework into which useful notational ideas have been placed.

The most common *file format* for Z uses the LaTeXmark-up defined by Spivey's *f*UZZ. This and other existing mark-ups/formats should be supported.

1.3 Sources of effort

The observation that core Z tools are neither likely to be commercially viable nor the subject of successful research proposals leaves the question of how they are to be paid for.

Two promising angles seem open:

- Advanced research projects may apply for a quantity of money to develop the necessary infrastructure for their real research. The drawback here is that research output is measured in most areas by the out-

put of learned papers, not by *completing* software. Something 'good enough' to support a proof-of-concept development, may not be good enough for a different project wanting different aspects of the core functionality.

- Many pieces of core functionality may be amenable to implementation by undergraduate (honours) or Master's students. Of course, many such projects happen already, but they often repeat the same work, and are almost never integrated together. There may be considerable educational value in, say, trying to develop a parser for Z, but some students could derive value by taking such a component from a shared repository.

Other ideas are sought.

If these tools are to be useful and re-usable in various ways, they must somehow fall within the broad thrust of 'free software'. The precise licencing terms will depend on the author, but the CZT initiative should expect mainly to encompass software which is free for almost any kind of use (other than sale for commercial gain).

The open-source model is not in itself sufficient to guarantee quality software development, though it offers a good start. Work should be subject to peer-review and widespread testing, so that it may be able both to support future student projects, but also be strong enough to be used industrially.

1.4 Implementation Technologies

One of the problems associated with academic tools projects is that they frequently use exotic implementation technologies which are themselves experimental. These may offer a rapid development route (the author's experience differs!), but those tools and languages are seldom supported or available in the medium-term.

Most of the world — including many commercial development environments — is a Microsoft Windows mono-culture. Many academics are committed to other platforms — Solaris, and Linux in particular. For wide applicability and maximum flexibility, the core of CZT must use readily available software tools, capable of running on all popular platforms. For this reason, the Java language will play a large part in CZT, though perhaps not an exclusive role.

1.5 Existing Tools

One valuable contribution to the community would be to develop Jonathan Bowen's list of Z tools. Someone could usefully catalogue which are 'live', which are available (free or at a price), the implementation languages, input and output formats, and capabilities of these tools.

2 Level 0: Interface Design

Abstract discussion of interfaces is hard at this point. The components identified in Level 1 might, however, usefully be abstracted as interfaces.

Moreover, CZT will need a detailed description of those file formats which are to be supported for interchange and general I/O.

3 Level 1: Infrastructure

Many separate components can be identified as worth implementing in their own right. Such a 'bottom-up' approach seems unusual, but may be suitable in this area, since the basic components are well-understood.

3.1 Internal Representation of structures

Several people have argued that the representation of Z terms as data structures is the key component of this type of tool-set. One requirement is clearly for a data structure which captures as precisely as possible the form of the specification under consideration. That is, it should reflect the precise structure of each input term, and also record any commentary present in the specification. This is, somehow, an *abstract* representation of the *concrete* syntax.

Various applications will need to manipulate this structure: it may gain annotations through type-checking or other processes; it may be rewritten using syntactic or logical re-writes, etc.

The ideal approach seems to me to be the creation of a collection of *interfaces* (or abstract classes?) which represent a specification and allow it to be manipulated using visitors etc. Classes which implement these interfaces may do more than necessary (storing extensive annotations, for example) without violating their ability to be used in simpler situations.

[Utting] CZT should produce accurate error messages. Where possible, error messages should include accurate references to the line and column of the SOURCE file which caused the error.

5a. Tools that transform the Z AST in various ways must try to preserve 'source locator' information which provides back-links to the source. For example, each AST object might have an optional attribute that records the source locator of that construct (which specifies the file name, line number and column number where the construct was originally read from).

5b. Any interchange formats must be able to include these 'locators'.

3.2 Input and output of specifications

Parsers are needed for a variety of mark-up and source formats, including as many as possible of those in current use. These include *f*UZZ-style LaTeX(or

oz-style); the email mark-up; one or more XML representations. Ideally, tools would also support a word-processed format such as RTF, since this is in widespread use.

The interfaces (and their implementations) described in section 3.1 can be used as the targets of such parsing activities.

Likewise, methods will be needed to output a specification stored in the internal data format (i.e. a format implementing at least the most basic interface), in a variety of textual formats, including those listed above.

Plainly, with just this collection of classes, a simple 'translation' tool with a command-line interface will be almost trivial to implement. For any given format, it should be capable as performing the identity translation.

3.3 GUI components

Classes consistent with the Swing library should be implemented, capable of instantiation to display a fragment or whole specification, as described in section 3.1. The simplest requirement would be to render a Z paragraph as a suitable subclass of `Container`. A fuller implementation would allow the display of smaller fragments (predicates, expressions); folding and unfolding lines; browsing of whole specification documents (with various material optionally elided); and so on.

A key aspect of this work will be to identify suitable fonts and unicode presentations of Z symbols. Portable and durable font-handling code will be needed, as will a distribution with at least one fully-capable font.

More elaborate GUI components can be envisaged which allow editing. Again, these should find an appropriate place within the Swing hierarchies. These might enable editing of full specifications, individual paragraphs, or smaller fragments — these will find application in general editing and also interaction with proof and animation tools.

3.4 Other components

The description in the other sub-sections has been rather Java-centric. Other components may usefully use different technologies. For example, some work has been undertaken with XML transformations; interplay with tools such as ZEUS (with FrameMaker) might also be relevant.

[Utting] To help tool builders who wish to use programming languages other than Java (the preferred language of CZT), such as Prolog or functional languages, a good approach is to provide interchange formats that can be easily read by that language. (Perhaps an XML format, and/or a Prolog-specific format etc.).

3.5 Enabling integration

CZT should be able to integrate with existing Z tools. It must be possible to connect it to existing interactive Z tools, like the CadiZ prover. It should also be possible to connect it to batch-oriented Z tools that produce reports or perform Z to Z transformations.

At this level, the need is for integration *components*; tools that will facilitate interchange via files. Additionally, the components should address issues of batch-invocation of external tools, and the scope for user interface integration.

As a research topic, the inclusion of Z fragments in XMI might usefully be investigated, with a view to interchange between CZT and popular UML tools.

4 Level 2: Approaching tools

4.1 A GUI

The components above might be combined to produce one or more CZT applications. The re-use of such components might enable different tools to adopt similar interfaces, for ease of learning.

An exemplar GUI would include:

1. a WYSIWIG editor for Z specifications
2. facilities to load specifications from file, save to file, and print
3. a type-checking facility (ideally pluggable), with error reporting linked to the editor

In order to pass beyond the mundane, it might also incorporate:

1. front-end support for animation of Z specifications — pluggable animator behind the scenes
2. support for re-writing expressions (calculation in Z)
3. support for full-blown proof tools

As many components of the GUI as possible would be readily replaceable, to provide a simple environment in which to demonstrate ideas and new projects.

The GUI tool should be capable of easy installation and use on at least MS-Windows and X-Windows (Solaris/Linux). A 'lite' version running as a Java applet is desirable, too.

4.2 Integration

[Utting]

2b. CZT must be able to invoke existing tools, so must have some knowledge of their command line options etc. For the batch-oriented tools that produce output files (perhaps Z files), CZT must also know roughly what the tool does and what kind of output it produces, so that the output can be incorporated back into CZT.

2c. Using existing tools means that CZT cannot have a completely uniform user interface. The core tools of CZT may follow a common GUI-style, but some add-on existing tools will continue to use their own interfaces.

A Terminology

[Utting]

Z source format The format of a source file that contains a Z specification plus natural language commentary, and possibly other things like diagrams, table of contents, indexes, specifications in other formal languages (statecharts etc.). This is the format that USERS create with some kind of editor.

Interchange format A file or storage format that is used to communicate a Z specification between tools. This may be used to communicate just the Z part of a specification, or all of the specification, so I propose two more specific terms:

Z interchange format An interchange format that contains ONLY the Z constructs of the specification. (The remaining parts, such as natural language, are discarded before creating this format.)

Complete interchange format An interchange format that contains EVERYTHING in the specification, including natural language, diagrams etc. as discussed above.

Annotation A piece of Z-related information that can be deduced automatically by some tool, and attached to a Z construct.

- Example 1: the inferred type of a Z function application. In the predicate, a = , the 'a' and '' might be annotated with P Colour, to indicate that a set of Colour values are being compared.
- Example 2: the inferred signature of a Z schema expression

There has been some debate about whether interchange formats should be annotated or not. The advantage of including annotations is that the receiving tool does not need to derive the annotations for itself (e.g., the type checking is already done). On the other hand, if the

receiving tool is not interested in that information, it can ignore the annotations. So I propose the following final four kinds of interchange format (and an abbreviation for each one):

Annotated Z interchange format (AnnZIF)

Non-annotated Z interchange format (ZIF)

Annotated complete interchange format (AnnCIF)

Non-annotated complete interchange format (CIF)

(If you use an abbreviation, please spell it out in full the first time you use it).

Note that, by definition, a Z source format is not 'annotated', because annotations are added automatically by tools. However, it is possible that a Z source format could also be used as an interchange format. (In that case, it would be a CIF).

AST 'Annotated Syntax Tree'. This is a Java data structure for representing Z constructs. It contains roughly the same kind of information as an Annotated Z interchange format (AnnZIF).

(NOTE: it would be possible to define four different kinds of syntax tree: non-annotated/annotated x complete/z-only. However, I think that the annotated, z-only one will be most common.

Source Locator a reference to a precise position in a Z source file (including file name, line number, column number etc.)