International Tables for Crystallography (2006). Vol. G, Section 5.3.2.1.1, pp. 499–500.

5.3. Syntactic utilities for CIF

BY B. MCMAHON

5.3.1. Introduction

Since the introduction of the Crystallographic Information File (CIF), the crystallographic community has produced a wide variety of tools and applications to handle CIFs. Many changes have been made to existing programs to input and output CIF data sets, and occasionally changes may have been made to internal crystallographic calculations to provide a better fit to the view of the data expressed by the standard CIF dictionaries. However, for most crystallographers with an involvement in programming, there is an understandable tendency to invest the minimum amount of effort needed to accommodate the new format. Their primary interest is in the understanding and discovery of the underlying physical model of a crystal structure.

This chapter reviews several general-purpose tools that have been developed for CIF to check, edit, extract or manipulate arbitrary data items, with little in the way of crystallographic computation. They are of interest to the end user who wishes to visualize a structure in three dimensions or submit an article to a journal but who does not want to be concerned about the details of CIF. They also include several utilities that are helpful for manipulating the contents of CIFs without the need to write a large and complex program. The programmer with an interest in writing complete and robust CIF applications should look at the comprehensive libraries described in Chapters 5.4 to 5.6.

Many of the programs described in this chapter operate purely at the syntactic level; they require no knowledge of the scientific meaning of the data items being manipulated. Others have some bearing on the *semantics* of the file contents, either explicitly through information about data types and interrelationships carried in external CIF dictionaries, or implicitly through the user's choice and deliberate manipulation of items based on an understanding of what they signify. Nevertheless, most utilities described here are characterized by an ability to handle CIFs of any content and provenance. The best example of a program able to handle arbitrary CIFs at a purely syntactic level is *Star_Base* (Spadaccini & Hall, 1994), described in Chapter 5.2.

It should be noted that not all the programs described here are fully compliant with the specification of Chapter 2.2, and others have implementation restrictions or known bugs. Many, especially the older programs, are no longer actively supported and need to be handled with care. However, they are included here for the record, and because they may provide useful ideas and suggestions to future developers in an area that can still accommodate a wider range of tools for different uses.

5.3.2. Syntax checker

A CIF must conform to a subset of the syntax rules of a general STAR File (Chapter 2.1), but with the additional restrictions and conventions described in Chapter 2.2. The syntax is rather simple and robust subroutines to create CIFs may easily be written by computer programmers. However, the use of ASCII character sets, deliberately expressive data names and simple layout conventions both permit and encourage users to edit the files with general text editors that cannot guarantee to retain syntactic integrity. Consequently, there is a definite use for a simple program that can check whether a file conforms to the specified syntax.

It is worth mentioning that programmable text editors such as *emacs* may be supplied with rules that can check syntax as a file is edited. A simple rule set (known as a *mode file*) has been developed (Winn, 1998) to indicate the different components of a CIF, as a first step towards a syntax-checking *emacs* mode.

The *Star.vim* utility of Section 5.2.4 provides a similar functionality for editing in the *vim* environment, although it is not capable of validation directly; nevertheless, the appearance of unexpected or irregular highlighted text can draw the user's attention to syntactic problems, a feature that is also useful in more extended editors such as *enCIFer* (Section 5.3.3.1).

5.3.2.1. *vcif*

A simple syntax checker for CIF is the program *vcif* (McMahon, 1998), which scans a text file and outputs informative messages about apparent errors. While conservative CIF parsing software will quit upon finding an error, *vcif* will attempt to read to the end of the file and list all clearly distinguished errors. However, its interpretation of errors depends on a close adherence to the CIF syntax specification and makes no assumption about the intended purpose of the character strings it reads. In consequence, a single logical error such as failing to terminate a multiple-line text string may cause the program to report many other apparent errors as it proceeds out of phase through the rest of the file.

5.3.2.1.1. *How to use vcif*

The program may be run under Unix or DOS by typing

```
vcif filename
```

where *filename* is the name of the file to test. If *filename* is given as the hyphen character -, the program will read standard input. Standard input will also be read if no file name is supplied; this allows the program to be used in a pipeline of commands.

A number of options may be supplied to the program to modify its behaviour. Without these options (*i.e.* invoked as above) a brief but informative message is written to the standard output channel for each occurrence of what the program perceives to be a syntax error.

For example, for the incorrect sample file of Fig. 5.3.2.1(*a*), the output is listed in Fig. 5.3.2.1(*b*).

Note that the sequence number of the line in which the error occurs is printed. The summary error message is output on a single line (longer lines have been wrapped and indented in Fig. 5.3.2.1 for legibility). Where the type of error necessarily affects only a single line, the program can recover and correctly identify errors on subsequent lines. Where possible, unexpected character strings are printed to help the user to identify the error. No attempt is made to assign any meaning to the data names or the data values in the

Affiliation: BRIAN MCMAHON, International Union of Crystallography, 5 Abbey Square, Chester CH1 2HU, England.

```
# Sample CIF with syntax errors

_date              'Monday 12 April 1999

_cell_length_a     7.514 (3)
_cell_length_b     9.467 (2)

    loop_
     _geom_bond_atom_site_label_1
     _geom_bond_atom_site_label_2
     _geom_bond_distance
                  O1   C2   1.342(4)
                  O1   C5   1.439 (3)

_example_comment
; The purpose of this example is to indicate how vcif
  describes some of the syntax errors it finds.
```

(*a*)

```
ERROR: No data block code before dataname at line 3
ERROR: Single-quoted character string does not
     terminate at line 4
ERROR: Unexpected string ((3)) at line 5
ERROR: Unexpected string ((2)) at line 6
ERROR: Number of loop elements not multiple of
     packetsize at line 15
ERROR: Text field at end of file does not terminate
```

(*b*)

Fig. 5.3.2.1. (*a*) An example CIF with a number of syntax errors and (*b*) the report of the errors produced by *vcif*.

file. Hence the same logical error (the detachment of a standard uncertainty in parentheses from its parent value) is indicated variously as an unexpected text string or as an extraneous loop item, depending on where it occurs in the file. Indeed, in the case of the incorrect number of loop elements, the program makes no attempt to identify which data value or values in the loop might be in error: it simply counts the number of values in a loop and complains when this is not a multiple of the number of data names declared in the loop header.

5.3.2.1.2. *Options to vcif*

A number of options may be supplied as command-line arguments to modify the output from *vcif*.

A more complete account is given of each error on its first occurrence when the program is invoked with the '*-v*' option. The output listing explains in more detail what the breach of syntax is and sometimes suggests how misunderstandings of the file structure result in such breaches (Fig. 5.3.2.2).

Each error message is prefaced by the word 'ERROR' (or occasionally another phrase such as 'WARNING' or 'STAR ERROR'). Three chevrons preface a printout of the beginning of the troublesome line. Then an expanded description of the error is given, prefaced by three asterisks, *on the first occurrence of each distinct error*. In this mode, only the first 20 errors are listed (the assumption is that this mode is best suited to novices, who should identify and correct each error in turn and would not want to be swamped by large numbers of error messages arising from a single error). More errors may be reported by using the '*-e*' command-line option.

The *quiet* option (*vcif -q*) outputs no error messages but instead returns to the calling environment an integer giving the total number of errors found. This option allows scripts or external programs to use *vcif* as a silent test of whether a file has any syntax errors.

```
ERROR: No data block code before dataname at line 3
>>> "_date"
*** A data block MUST begin with a data_something
      declaration.
ERROR: Single-quoted character string does not
      terminate at line 4
>>> "_cell_length_a"
*** The indicated line appears to contain some word
      or words introduced by a single quote, but not
      terminated with a matching single quote.
ERROR: Unexpected string ((3)) at line 5
*** There is an unexpected word or number as
      indicated. This may be because a loop is
      intended but the loop_ keyword has been missed
      out; or a phrase with several words is not
      enclosed in matching delimiting quote marks; or
      a text field (extending over several lines) is
      not properly closed with a final semicolon; or
      a data_ block header has not yet been seen.
ERROR: Unexpected string ((2)) at line 6
>>> "_cell_length_b"
ERROR: Number of loop elements not multiple of
      packetsize at line 15
>>> "_example_comment"
*** A loop_ header defines a list of datanames. The
      values following this header are assigned in
      sequence with the datanames in the header, so
      each packet of information (or row in the table
      of values defined by the loop structure) must
      have the same number of values as there are
      datanames declared in the loop header. Common
      reasons for this error include: omission of a
      value where the associated data are absent
      (insert . or ? as placeholders); numeric values
      where the standard uncertainty (or e.s.d) has
      come adrift from its associated value (e.g.
      10.925 (2)); multi-word phrases or text entries
      that are not properly delimited with quote
      marks or initial semicolons.
ERROR: Text field at end of file does not terminate
>>> ""
*** Is the CIF complete?
```

Fig. 5.3.2.2. Verbose error listing from *vcif* when run with the '*-v*' option on the example of Fig. 5.3.2.1.

A related option, *vcif -b*, counts errors and returns the result as an integer to the calling environment, as in the previous case; but additionally outputs a list of all the data-block codes in the file. While adding nothing to the syntax-checking function of the program, this provides a useful small utility for simply listing data-block names.

Although intended for use with the restricted STAR File syntax permitted for CIF (Chapter 2.2), *vcif* may also be used with the '*-s*' option to check the syntax of CIF dictionary files, which may include save frames. The program does not, however, handle nested loop structures.

The program will flag as an error any line of greater than 80 characters length (the original limit in the CIF version 1.0 specification; see Chapter 2.2), but this behaviour may be overridden with the '*-l*' option. If used, only lines longer than the specified number of characters will be reported and the reports of such lines will be prefaced with the word 'WARNING'. Likewise, the '*-w*' option may be used to override the CIF version 1.0 restriction of data names and data-block codes to 32 characters.

Other options allow the program to write extensive debugging information to a user-specified file, indicating its internal state upon processing each token of input, and to list either a brief summary of how it may be used or its current version number.