International Tables for Crystallography (2006). Vol. G, Chapter 5.3, pp. 499–525.

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.

5.3.2.1.3. *Limitations of vcif*

Because the program is testing certain properties of character strings within logical lines of a file, it stores a line at a time for further internal processing. If a line contains a null character (an ASCII character with integer value zero), this will be taken as the termination of the string currently being processed, according to the normal conventions in the C programming language for marking the end of a text string. In this case, subsequent error messages may not reflect the real problem. The null character, of course, is not allowed in a CIF.

vcif also interprets syntax rules literally, so a misplaced semicolon might mean that a large section of the file is regarded as a text field and too many or too few error messages are generated. This can make a correct interpretation of the causative errors difficult for a novice user.

5.3.3. Editors with graphical user interfaces

A useful class of editing tool is the graphical editor, where different types of access can be provided through icons, windows or frames, menus and other graphical representations. The availability of standard instructions through drop-down menus makes such tools particularly suitable for users who are not expert on the fine details of the file format. The ability within the program to restrict access to particular regions of the file makes it easier to modify the contents of a CIF without breaking the syntax rules. A small but growing number of such editors are becoming available, such as those described here.

5.3.3.1. *enCIFer*

The program *enCIFer* (Allen *et al.*, 2004) has been developed as a graphical utility designed to indicate clearly to a novice user where errors are present in a CIF, to permit interactive editing and revalidation of the file, and to allow visualization of three-dimensional structures described in the file. In its early releases, it was targeted at the community of small-molecule crystallographers interested in publishing structures or depositing them directly in a structure database. Version 1.0 depended on a compiled version of the CIF core dictionary, but subsequent versions allow external CIF dictionaries to be imported. At the time of publication (2005), development is concentrating on support for DDL1 dictionaries.

Given its target user base, the purpose of the program is to permit the following operations within single- or multi-block CIFs:

(i) Location and reporting of syntax and/or format violations using the current CIF dictionary.

(ii) Correction of these syntax and/or format violations.

(iii) Editing of existing individual data items or looped data items.

(iv) Addition of new individual data items or looped data items.

(v) Addition of some standard additional information *via* two data-entry utilities prompting the user for required input ('wizards'): the *publication wizard*, for entering the basic bibliographic information required by most journals and databases that accept CIFs for publication or deposition; and the *chemical and crystal data wizard*, for entering chemical and physical property information in a CIF for publication in a journal or deposition in a database.

(vi) Visualization of the structure(s) in the CIF.

In all cases where data are edited or added, *enCIFer* can be used to check the format integrity of the amended file.

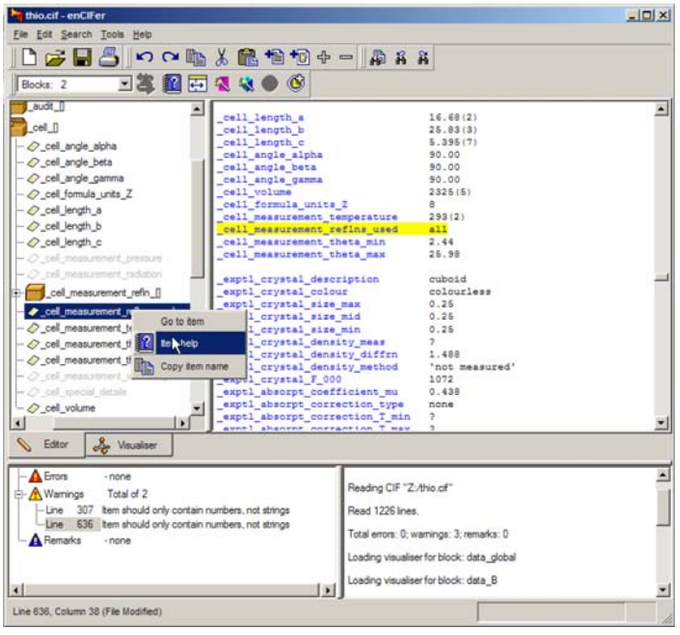

Fig. 5.3.3.1. The *enCIFer* graphical user interface.

5.3.3.1.1. *The main graphical window*

Fig. 5.3.3.1 is an example of the use of *enCIFer* to read and modify a CIF. The figure shows the components of the main window after a file has been opened. Beneath the standard toolbar that provides access to operating-system utilities and to the main functions of the program itself is a task bar (here split over two lines) providing rapid access to a subset of the program's features. Under this are two large panes. The pane on the right is the editing window, where the content of the CIF is displayed and may be modified. The left-hand pane is a user-selectable view by category of the data names stored in the CIF dictionary against which the file is to be validated. At the bottom are two smaller panes. The one on the right logs the session activities and displays informational messages. The left-hand pane lists errors and warning notices generated by the validation system. Errors are labelled by line number, and selection of a specific message (by a mouse double-click) scrolls the content of the main text-editing window to that line number.

Tabs in the middle of the display allow the user to switch rapidly between the editing mode and a visualization of the three-dimensional structures described in the CIF.

These components are described more fully below, followed by a description of the other windows that may be created by a user: the help viewer, the loop editor and the data-entry wizards.

5.3.3.1.2. *The interface toolbar*

This toolbar provides menus labelled 'File', 'Edit', 'Search', 'Tools' and 'Help' that provide the expected functionality of graphical interfaces: the ability to open, close and save files, store a list of recently accessed files, spawn help and other windows, allow searching for strings within the document, allow the user to modify aspects of the behaviour or the look and feel of the program, and provide entry points for specific modes of operation. The most useful of these utilities can also be accessed from icons on the task bar. They are discussed in more detail in the following section.

This main menu is structured in a way familar to users of popular applications designed for the Microsoft Windows operating

system, although the *enCIFer* program runs on a variety of different operating systems and machine hardware platforms. Nevertheless, the use of a common menu style makes the initial use of the program much easier for novice users and allows the program to be effectively used without detailed study of its documentation.

5.3.3.1.3. *The task bar*

The task bar allows rapid one-click access to the standard operations of creating a new document, opening, saving or printing the contents of the current file, copying, cutting and pasting text, searching for specific text within the document, and undoing or redoing previous edits.

Two buttons allow insertion of complete text files. One allows the user to select any file from local or network-mounted file systems. The other imports a specific file (the location of which may be specified by the user through the 'Preferences...' selection of the main 'Edit' menu). While this specific file may contain anything, it is intended to be a template CIF that a user will tailor to meet their own requirements. The default provided with the software is a standard template distributed by the IUCr for use in submitting articles to *Acta Crystallographica*. In either case, the file is imported at the current editing location and is not subject to validation upon input; the user must manually revalidate the file after import.

An icon on the task bar allows the user to run a validation procedure. This icon will be dimmed (indicating that the validation procedure may *not* be run) unless the user has modified the contents of the CIF. Other icons on the task bar behave in the same manner, allowing the procedures with which they are associated to be executed only under appropriate circumstances. Thus, for example, the looped list editor is not invoked unless the user clicks within the reserved word `loop_` in a list header.

Similarly, the 'help' icon in the task bar is dimmed unless the user has selected a data name in the CIF; when this is done, the icon is activated and clicking on it launches a help window containing the CIF dictionary definition of the data item.

The task bar also contains a drop-down menu listing all the data-block names in the current file. When the user selects one of the data-block names, the edit cursor is positioned at the head of the matching data block in the edit window. This is a rapid and efficient way of navigating within large and complex files.

The other buttons provided on the task bar allow the user to: reduce or increase the font size in the editing window; create a new looped list within the loop-editing window; invoke the publication and data-entry wizards; and hide or reveal the dictionary browse window pane.

Users may modify the appearance of the task bar to retain or conceal subsets of these icons, depending on which they find most useful.

5.3.3.1.4. *The main edit pane*

The main edit pane is a text-editing area where the user may directly modify the content of a CIF. Colours and font styles are used to indicate different syntactic elements. The details of the colours and styles may be modified to suit the user.

For the novice user, this is perhaps the most immediately helpful feature offered by this program. When a trailing semicolon is inadvertently lost from an extended text field, typical sequential parsers may interpret succeeding tokens as part of the quoted text and produce misleading error reports. Within the *enCIFer* edit window, all such text is marked up in a specific colour (green by default) so that the fault is much more obvious to the human eye and its source much easier to locate.

Two other typographic cues are used to help the user to trace errors, or to ensure that certain text has been input correctly. Subscripts and superscripts are represented in a smaller typeface (and in a different colour) so that missing delimiter characters are again obvious to the eye. Secondly, some special characters in the conventional CIF encoding (such as Greek letters) are displayed in an appropriate symbol font when the file is first loaded, so that for example the input string `\a` is rendered as α. Note that the backslash character is retained, and that the symbol character is not generated as new text is input or edited. This scheme therefore has some potential for confusion, but is nevertheless helpful in checking that less obvious special codes have been entered correctly.

The user is free to enter arbitrary text in this pane, possibly breaking CIF syntax rules in the process. Only when the revalidation process is manually invoked will the file be rescanned and any errors reported.

5.3.3.1.5. *The dictionary browse pane*

The upper left-hand pane in Fig. 5.3.3.1 illustrates the dictionary browser, an optional graphical view of the contents of the CIF dictionary against which the file is being validated. (The presence or absence of this pane is toggled from an icon in the task bar.) Box icons represent the contents of categories, and the tree of category containers may be expanded or collapsed as desired to show individual items within categories.

A dictionary view is generated for each separate data block in the CIF. Within the dictionary view of an individual data block, those data items present in the data block are shown in bold; other items defined in the dictionary but absent from the current data block appear in a lighter colour.

Within the dictionary browse pane, a user may select (with a click of the appropriate mouse button) a menu of three options which depend on whether the data name is present or absent in the data block. If present, one option positions the cursor in the editing window at the location of the selected data item. If the item is absent from the data block, the user is given the option to paste the data name into the editing window at the current insertion point. The other options (in both cases) are to copy the data name to the clipboard or to open the help window with the CIF dictionary definition of the selected item.

5.3.3.1.6. *The error notification pane and logging area*

The lower left-hand pane of Fig. 5.3.3.1 illustrates typical error notices generated by the parser when the validation process is invoked. At present, the classification of the severity of errors is guided by the editorial requirements of databases and journals, and does not necessarily match the formal errors dictated by the CIF specification. It is likely that this will change in future releases as validation is driven increasingly by the dictionaries rather than by hard-coded subroutines.

A convenient feature is that double-clicking on the line number in the error report relocates the cursor to that line in the editing pane. At present, error messages are listed by line only – they are not grouped by data block.

The user has a small number of options to control error notification. The choice of the maximum number of consecutive error lines to permit before error checking is abandoned is a useful way, especially for novices, to reduce the amount of output generated by severe syntax errors and to focus on repairing individual errors. The user may also specify a file that contains a set of CIF data names which are considered *mandatory* components of a particular file. Absence of any of these items from the current data

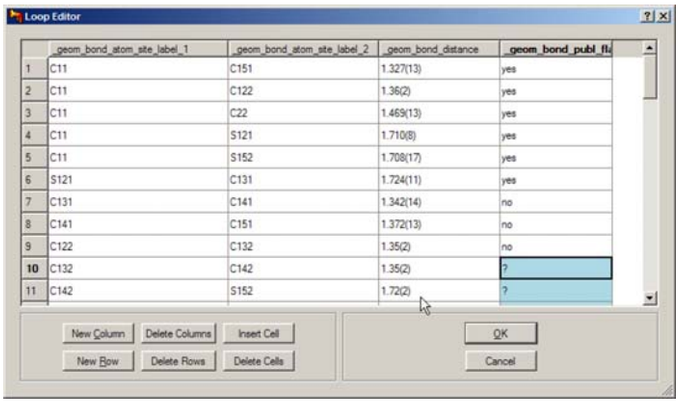

Fig. 5.3.3.2. The *enCIFer* loop editor.

block is flagged as an error. The program log in the lower right-hand part of the program window records the history of the user's interactions with the file during the current editing session.

Information is written to the status bar (the lower margin of the window) to indicate the location by line and column number of the editing cursor.

5.3.3.1.7. *The loop editor*

The program has a useful spreadsheet-style editor for looped lists (Fig. 5.3.3.2). A particular benefit of this style of display is that the spreadsheet cells are arranged in a rectangular grid, so that visual scans can often detect deviations from a pattern of values within a column, thus making it easy to identify placement errors where values have been omitted or inadvertently conjoined. Such errors are not always obvious by direct visual inspection of a CIF, where the layout of a looped list need not follow any regular pattern.

The buttons to add or delete columns allow for the straightforward addition or deletion of data items from the loop. If the user selects the 'New Column' button, a small pop-up window helpfully provides a view of the associated dictionary (in the same hierarchical category-based tree view of the dictionary browser pane) to help the user select the required new data name. The 'Insert Cell' and 'Delete Cells' buttons are convenient tools for the realignment of rows and columns where values have been omitted or misplaced.

The loop editor is invoked from one of two buttons in the task bar, allowing either the creation of a new looped list or the modification of an existing one. As with the application as a whole, there is no dynamic validation of input; the new list must be saved and the entire CIF then manually revalidated.

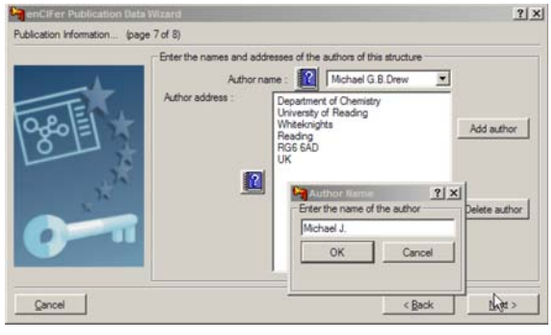

Fig. 5.3.3.3. The *enCIFer* publication data wizard. Information about the title and authors of an article to be submitted for publication is requested through a sequence of linked dialogue boxes.

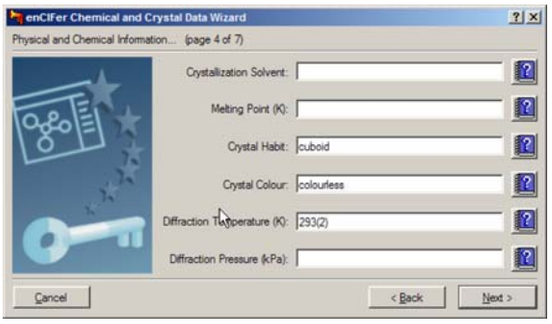

Fig. 5.3.3.4. The *enCIFer* chemical and crystal data wizard.

5.3.3.1.8. *The publication and chemical and crystal data wizards*

The user may invoke data-entry 'wizards', subordinate programs that prompt for particular data items useful for the publication of a crystal structure report or for the deposition of a crystal structure in a database. This is the kind of information that might be requested in the *Notes for authors* for a journal, and it is helpful if the information is routinely requested from inexperienced authors during normal use of the software. The data-entry tools are known as 'wizards' because they will utilize information already in the file.

Hence, as shown in Fig. 5.3.3.3, details of an article's contact author are retrieved from the CIF and used to seed a list of contributing authors. As the address for each author is entered, the program makes each new address available as a stored record for easier input of additional information.

Fig. 5.3.3.4 demonstrates the same approach to encouraging authors to supplement information already in the CIF with related chemical (or crystal) data not usually provided by the CIF generators embedded in crystallographic structure determination programs.

5.3.3.1.9. *The visualization window*

A final useful feature of *enCIFer* is its ability to visualize the three-dimensional structure of molecules described in the data blocks of a CIF. Fig. 5.3.3.5 demonstrates crystal packing with

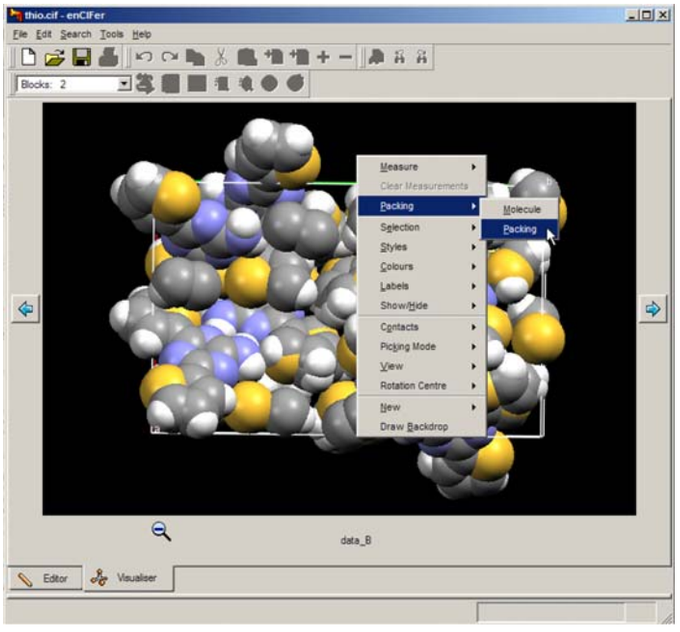

Fig. 5.3.3.5. Visualization of a molecular and crystal three-dimensional structure with *enCIFer*.

a space-filling molecular representation, and the drop-down menu indicates some of the options available to modify the appearance of the graphics. The molecular-graphics library used by the program is part of the larger database interface software package developed at the Cambridge Crystallographic Data Centre. In the present version, the visualizer is run only upon initial parsing of the input CIF, and therefore does not provide an ability to track visually the molecular changes associated with direct modification of the contents of the file.

5.3.3.2. *CIFEDIT*

The *CIFEDIT* program (Toby, 2003) is written in Tcl/Tk (Ousterhout, 1994) and provides an application for viewing and editing CIFs. The code is written in such a way that it can be embedded into larger programs to provide a CIF-editing interface within larger application suites.

The current version of the program is able to validate CIFs against both DDL1 and DDL2 dictionaries, although the DDL2 validation is currently less complete than for DDL1. For example, numeric values are checked against permitted enumeration ranges only for DDL1. Dictionaries are accessed through index files, each of which contains Tcl data structures that point to the location of the definitions in the dictionary file itself and store information such as units and enumeration ranges that can be used for data validation. A utility provided with the program allows a user to generate new index files when new versions of the dictionaries become available. It is intended that dictionary indexing will be incorporated within the main application in the next program release, so that interactive dictionary selection will be possible.

When a CIF is opened, the contents are parsed and validated against one or more user-selected dictionaries. Errors are displayed in a pop-up window and may be written to a file or viewed within the application. The main program window displays the contents of the CIF in two primary panes (Fig. 5.3.3.6). In the left-hand pane, a tree structure shows the data blocks in the file and the data names present in each block. The data blocks may be expanded or collapsed by the user, to present an overview or a detailed view of the data structure of the file. Underneath the icon representing the data block, non-looped data items are listed alphabetically. The figure demonstrates how a single value may be selected in the left-hand pane (`_cell_length_a`) and displayed in the main window. Physical units for the selected quantity are extracted from the corresponding dictionary definition and presented alongside the numeric value. The dictionary definition may also be displayed in a separate pop-up window using the 'Show CIF Definitions' button.

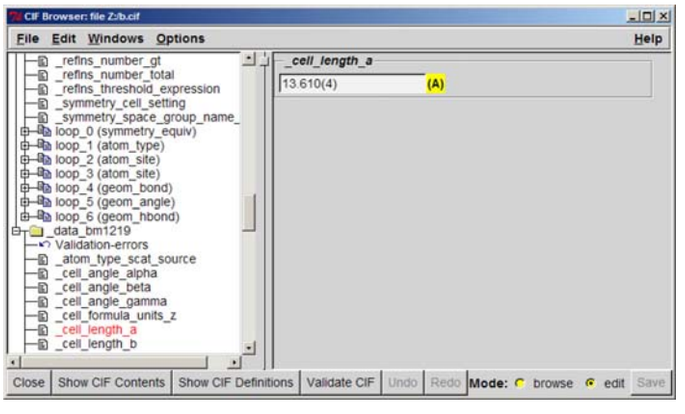

Fig. 5.3.3.6. The use of *CIFEDIT* to display and alter the contents of a CIF; here a non-looped data item is shown.

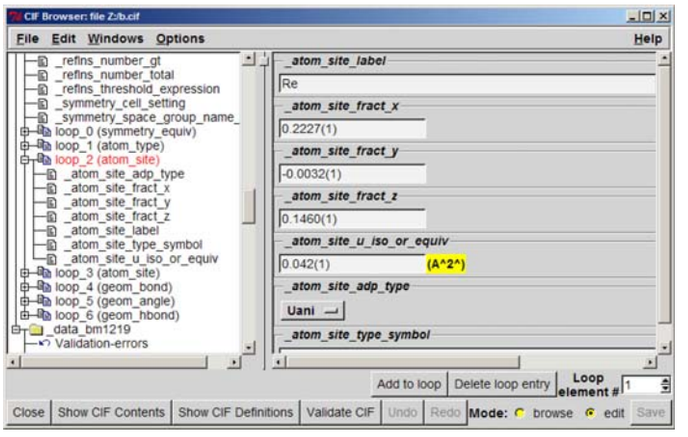

Fig. 5.3.3.7. Row-based loop editing with *CIFEDIT*; here loop_2 (comprising the ATOM_SITE category) has been selected by the user; the editing cursor begins at row 1 of the loop.

The program may be run in two modes: a 'browse' mode, where the selected value is displayed in the main pane, but may not be altered; and an 'edit' mode (as in the example) where the value appears in an editable text widget.

Data loops in the CIF are displayed after the alphabetical list of non-looped items. The loops are numbered sequentially from zero and an indication of the loop category is given in parentheses in the tree-view window. The loop 'branches' of the tree may be expanded or collapsed as the user wishes.

Loops may be viewed and edited in two ways: by row or by column. If the user selects the loop title node in the hierarchical view pane, the loop is presented by row, starting in sequence at row 1 (Fig. 5.3.3.7). Other rows may be selected by using the address box in the lower-right-hand part of the window. Alternatively, if the user selects an individual data name within the loop representation in the hierarchical view, all instances of that data item within the loop are displayed in the main pane. (In practice the number of values shown is constrained to a maximum number that the user may choose, so that the application does not run out of memory if there are very large loops.)

For items with a restricted set of permitted values in the dictionary, the editing function allows the user to select only one of the permitted options *via* a drop-down menu.

While the application is intended to be used in this structured and itemized mode, there is an option to open the entire CIF in a text-editing window if there are errors that cannot be handled in the normal mode. This is not recommended, but is occasionally convenient. While this free-text editing mode is in operation, the ability to modify the file through the structured editing pane is suspended to avoid conflicting changes.

After any change has been made, the user may revalidate the file. This is strongly recommended after making changes in the free-text editing mode.

5.3.3.3. *HICCuP*

The program *HICCuP* (Edgington, 1997) was an early graphical utility developed at the Cambridge Crystallographic Data Centre for interactive editing and validation of a CIF. It is no longer supported, having been replaced by *enCIFer* (Section 5.3.3.1). Nevertheless, it contained some interesting features and is of potential interest to developers using multiple-platform scripting languages. It was implemented in the Python language (van Rossum, 1991) and required that Tcl/Tk (Ousterhout, 1994) be also available on the host computer. The name of the program is an acronym for 'High-Integrity CIF Checking using Python'.

HICCuP was designed to allow users of the Cambridge Structural Database (Allen, 2002) to check structures intended for deposition in the database and therefore included a range of additional content checks specific to this purpose. These could, however, be disabled by the user.

5.3.3.3.1. *Interactive use of the program*

5.3.3.3.1.1. *The control window*

Because *HICCuP* was designed as an interactive tool, upon invocation it presented to the user a *control window* from which CIFs could be selected for analysis and in which summary results of the program's operations were logged. Fig. 5.3.3.8 shows an example of the control window after a single CIF has been loaded.

In the large frame below the file-entry field are listed the data blocks found by the program. The names are highlighted in various colours according to the highest level of severity of errors found within the corresponding data block.

Because the utility was designed for processing large amounts of CIF data for structural databases, it was considered useful to supply a compact visual indicator of the progress of the program through a large file. This takes the form of a grid of rectangular cells, one column for each data block present. Each column contains three cells, which monitor the performance of checks on the file syntax, conformance against a CIF dictionary, and other checks specific to the requirements of the Cambridge Crystallographic Data Centre. As each data block was checked, the corresponding cells were coloured according to the types of error found. Different colours were used to indicate: no errors; structure errors in the initial syntax tests; dictionary errors; or a deviation from certain conventions used by journals and databases in naming datablocks.

The large frame at the bottom of the control window provides a text summary of the same information, listing the number of errors found.

Check boxes and an 'Options…' button allowed some configurability of checks by the user.

5.3.3.3.1.2. *The report frame and edit window*

The user could get more details of the reported errors by clicking on the name of the data block of interest in the control window. The text of the CIF would appear in a new window positioned at the point where the program has detected the first error and a terse statement of the type of error, with a longer explanation of its nature and possible cause, would be given.

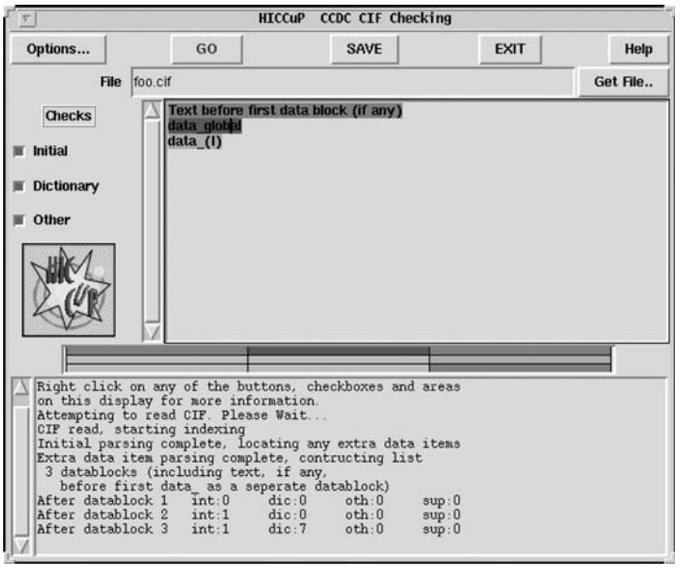

Fig. 5.3.3.8. Control window of the *HICCuP* application.

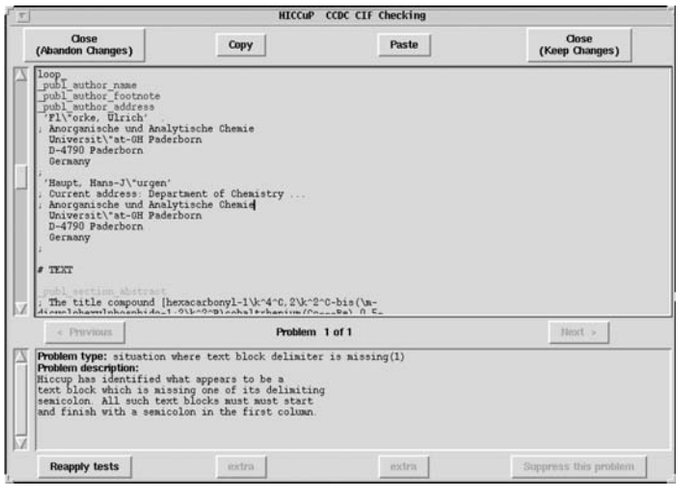

Fig. 5.3.3.9. *HICCuP* edit window and error description.

In the example of Fig. 5.3.3.9, the program has detected that there is a missing text delimiter (a semicolon character), and positions the text in the upper frame at the likely location of the error. The program has attempted to localize the region where the error may have occurred. Because a text field might contain arbitrary contents, including extracts of CIF content, it is impossible to be sure on purely syntactic grounds of the nature of the error. Nonetheless, some heuristic rules serve to identify the author's likely intent in the majority of cases. So, in this example, the user may scan the file contents in the vicinity of the line highlighted by the program and find the error within a few lines (in this example an incorrectly terminated `_publ_author_footnote` entry beginning 'Current address:').

For this example, the more literal *vcif* error analysis provides only the message

```
ERROR: Text field at end of file does not terminate
```

The upper frame in this window is an editable window, so that the user could modify the text and revalidate the current data block. Only when a satisfactorily 'clean' data block was obtained were the changes saved, and the modified data block written back into the original file.

5.3.3.3.1.3. *Dictionary browsing*

An additional useful feature of the program was its interactive link to a CIF dictionary file (Fig. 5.3.3.10). The browser window contains the definition section of the dictionary referring to

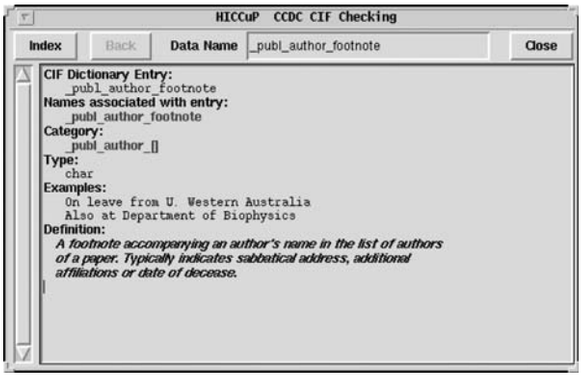

Fig. 5.3.3.10. *HICCuP* dictionary browser window.

the selected data name and hyperlinks to definitions of other data names referred to. Additionally, there is a small text-entry box allowing a specific definition to be retrieved and an 'Index' button to list all available definitions.

5.3.3.3.2. *Options*

As already mentioned, the user could modify the detailed mode of operation of the program. Any or all of the 'initial', 'dictionary' or 'other' checks could disabled.

The 'dictionary' checks could be modified by the user through the 'Options' button of the main control window. The CIF dictionary for validation could be specified; the dictionary itself had to be translated from a source file in DDL format to a Python data structure.

The types of dictionary-based validation supported by the program were:

(i) *List Status* (checking whether a data value should be included in a looped list),

(ii) *Limited Enumeration Options* (checking that a data value is one of the permitted codes where such a constraint exists),

(iii) *Incorrect Enumeration Case* [a special case of (ii), where a data value matches a permitted code except for incorrect alphanumeric case],

(iv) *Enumeration Range* (the data value falls outside the range permitted),

(v) *Value Type (numb or char)* (the data value has the wrong type),

(vi) *List Link Parent* (a data item is present within the data block, but its mandated parent item is not – for example, the data item `_atom_site_aniso_label` should not be present without its parent data item `_atom_site_label`),

(vii) *List Reference* (the required data name used to reference the loop in which the current data name appears is missing),

(viii) *Esd Allowable* (a data value appears to have a standard uncertainty value where one is not expected).

The user could also supply the program with a list of data names that do not appear in the validation dictionary but for which no warning message should be raised. The program normally flagged such nonstandard data names as possible errors and suggested the possible form of a standard data name that might have been intended. This was useful in catching misspellings of additional data items entered by hand.

The program could also be run in a batch mode when the objective was to work through a large volume of CIF data and identify the data blocks that require attention. This mode of operation is particularly useful in databases or publishing houses. In this mode, input is from a named file or from the standard input channel; output is written to standard output or redirected to a results file. The operation of the program may be controlled by the application of various command-line flags.

5.3.3.4. Platform-specific editors

As well as the tools described earlier in this section, which are designed to run under a variety of common operating systems, there are some applications restricted to users of particular types of computer. Here we mention two that run in the popular Microsoft Windows environment on personal computers.

5.3.3.4.1. *beCIF*

The Windows program *beCIF* (Brown *et al.*, 2004) is still in prototype. It is a DDL1-dictionary-driven CIF manipulation tool that does not require detailed knowledge of CIF or dictionary

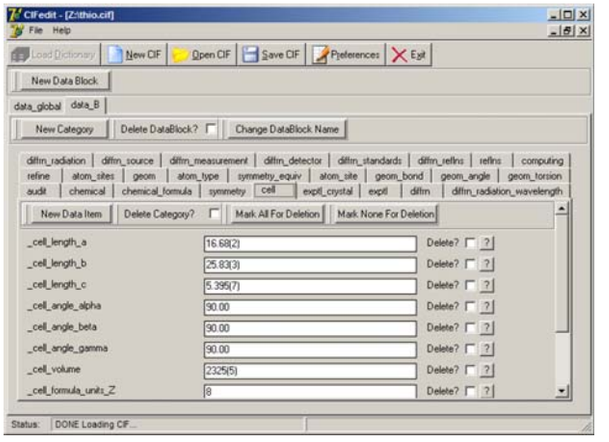

Fig. 5.3.3.11. A category view in the *beCIF* editor of a CIF with navigation by tabs.

structures. It provides a rather different view of the contents of a CIF from the applications discussed above through an interface that will be familiar to users of Microsoft Windows applications. When the application is opened, the user is prompted to provide the location of a CIF dictionary (at any one time, only a single dictionary file may be loaded). This dictionary is loaded into memory and used to validate CIFs upon input. As a data file is read, discrepancies from the types and value ranges permitted by the dictionary are listed in an information window.

The file contents are presented in a number of panels, one per dictionary category, between which the user may navigate by selecting the tab with the desired category name (Fig. 5.3.3.11).

At the highest level, tabs allow the user to choose the data block of interest. Buttons are provided to delete a data block entirely, to rename it or to create a new data block.

Within each data block, the user may add new categories. Again, to help the novice user, when the button 'New Category' is selected, a list of only those categories described in the current dictionary but absent from the current data block is presented to the user. Each category present in the data file is accessed through its own tabbed display panel.

Where the category contains non-looped data items, values may be edited within individual text widgets; data items may be removed by selecting the adjacent check box; or new data items may be added by selecting the 'New Data Item' button to create a dialogue box offering a choice of the remaining data items in the dictionary category. Against each data item a button provides access to a pop-up window containing the relevant dictionary definition.

For a category with looped data, the contents are displayed in a spreadsheet-style representation, with columns headed by the matching data name and rows numbered for convenience (Fig. 5.3.3.12).

The changes requested to the CIF are only effected when the user selects the 'Save CIF' button. Unlike many other of the CIF editors previously discussed, this program does not make any effort to retain the initial ordering of the input data, nor does it preserve comments. The edited CIF may therefore be superficially very different from the input file; however, the only significant differences in content will be those introduced through use of the editing functions within the application.

5.3.3.4.2. *printCIF for Word*

The tools described so far emphasize the data content of a CIF. *printCIF for Word* (Westrip, 2004), on the other hand, was commissioned to help prospective authors of structure reports in the

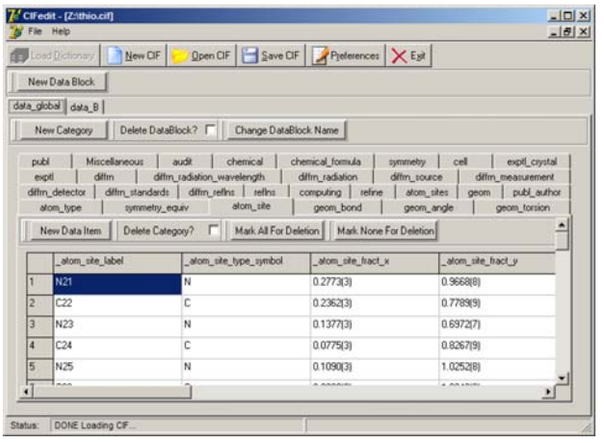

Fig. 5.3.3.12. Representation by the *beCIF* editor of looped data within a category (here ATOM_SITE) in spreadsheet style.

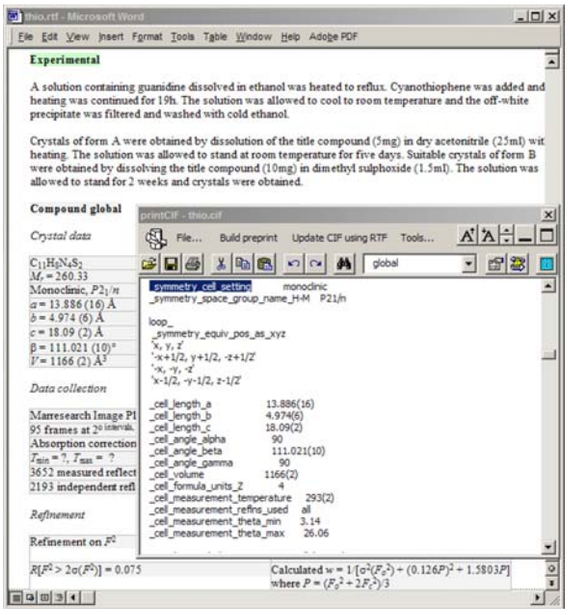

Fig. 5.3.3.13. The dual RTF/CIF editing windows in the *printCIF for Word* application. In this example, the author has selected the word 'Monoclinic' in the read-only table of crystal data; the corresponding CIF data item **`_symmetry_cell_setting`** is highlighted in the CIF window, where it may be edited.

IUCr journals to visualize and prepare for publication complete papers submitted in CIF format. Chapter 5.7 describes the workflow and processing of such submissions. Here is given a brief description of the use of the *printCIF* software from an author's viewpoint.

This application also differs from others discussed in this chapter in that it is rather specific to a particular program environment, being written as Visual Basic macros embedded in a Microsoft *Word* template document. Efforts are under way to provide versions that can run with other word processors. Nevertheless, *Word* is currently sufficiently widespread that the utility is likely to be of use to a large community.

Typically the author begins by double-clicking on the icon associated with the printcif.dot template file. The initial macros are loaded and the author is prompted to provide the location of a CIF. As the CIF is imported into the application, the data items that will be used in the publication are extracted and converted into a rich-text format (RTF) representation. For extended text fields, this RTF content may be edited directly in the word-processing environment; this makes it easy for authors to compose and edit continuous text in a familiar way. Numeric and brief textual data items from the CIF are processed and presented in read-only fields in the manner in which they will appear in the journal, often as entries in a table or as a list of brief experimental details. These fields may not be edited within the RTF representation; if it is necessary to change these, the author must modify the data value in the CIF itself. To assist the author, the contents of the CIF are opened in a text-editor window alongside the formatted representation. The CIF and RTF representations are linked; if the author selects text in the RTF window, the corresponding CIF data item is highlighted within the text-editor window (Fig. 5.3.3.13).

The advantages to the author of editing in RTF format are that existing text may be cut and pasted from other applications, and formatting features, such as subscript or superscript text, Greek letters and other special symbols, may be entered through the word-processor's menu-driven interface, rather than by use of the rather unmemorable ASCII codings used in CIF.

The major disadvantage is the need to recognize that two versions of the file, both editable, are accessible at the same time; and care must therefore be taken to ensure that conflicting changes are not made, and that the author is aware of which version is currently the master. The function 'Update CIF using RTF' (in the toolbar of the CIF editing window) will reimport into the CIF all the editable content from the RTF window, replacing any existing data items. The complementary function, 'Build preprint', creates a fresh copy of the preprint representation of the document in RTF format.

A number of options are available to modify the preprint that is generated (for example, by printing a complete list of the geometry included in the CIF rather than just the items flagged for publication; or listing the atomic coordinate data). The general style is that of *Acta Crystallographica Section C* and *Section E*; nevertheless, the application may be useful to users who do not intend to submit to these journals but who wish to produce an attractive representation of the content of their CIFs.

Utilities are provided to create tables in the RTF environment suitable for embedding in the CIF, to browse the contents of the CIF core dictionary and to validate the syntax of the CIF. The application is not dictionary-driven, however, and does not carry out detailed consistency checks. It is therefore best considered as an aid to publication, to be used alongside data-centric editors and validation tools such as *enCIFer*.

A particularly useful self-documenting feature of *printCIF for Word* is that the User Guide is automatically opened when the application is started, before a CIF is loaded.

5.3.4. Data-name validation

In a CIF, a data name (a character token beginning with an underscore character, _) is an essential handle on an item of data within a data block. Equipped only with knowledge of the data names appearing in a CIF, a user may extract, reorder or query the information content of the file. Such manipulations require no prior knowledge of the semantic content of the data. However, for most practical applications it is important to know the meaning attached to data names, and CIF dictionaries provide the mechanism for associating a data name with its intended meaning for an application. It is therefore valuable to be able to check whether data names in a CIF match those defined in a dictionary file. It is also valuable to check the consistency of the data names listed in the dictionary file itself; since this will be used by external applications to validate data names, it is essential that it be internally consistent.

Hence there is a real need for a utility to validate data *names* – effectively a CIF spelling checker.

5.3.4.1. *CYCLOPS*

The program *CYCLOPS* (Hall, 1993; Bernstein & Hall, 1998) was written specifically to address the problem of validating CIF data names. Its use extends beyond simply identifying data names in a CIF data file and checking that they are defined in a dictionary. Any ASCII file may be input, allowing for the checking of CIF data names in any text documents or program source.

The program was originally written in Fortran as an aid to ensuring that the original core CIF dictionary was free from data-name errors; subsequently it was extended to be able to read multiple dictionaries in DDL1 and DDL2 formats, and to resolve data-name aliases across multiple dictionaries. The extended version was written with the library routines of the *CIFtbx* toolkit (Hall & Bernstein, 1996) described in Chapter 5.4 and is distributed as an example application with *CIFtbx*. The description below refers to this extended version, also known as *CYCLOPS2*.

5.3.4.1.1. *Operation*

The program determines the dictionary (or list of dictionaries) against which to validate the input text file (see below for the method of passing such information to the program). It opens each dictionary in turn and stores all data names defined in the dictionaries. Where the same name is defined in multiple dictionaries, the behaviour is determined by a command-line switch.

The text file is then input and parsed for candidate data names. Because the program is designed to check potential data names embedded in ordinary text files, it is not sufficient to apply the CIF parsing rule of a white-space-delimited character string beginning with an underscore character. Instead, character strings are sought that begin with an underscore optionally preceded by white space or one of the characters `,.([{</\|'":*` and followed by white space, one of the characters `,.)]}>/\|'"-=?!;:` or by the end of a line.

For each candidate data name found in this way, matching data names in the stored list are identified in one of three ways:

(i) If the data name is not preceded by the asterisk character `*` and it does not end with the underscore character _, then search for an identical match.

(ii) If the data name ends with the underscore character _, then search for a match in the dictionary where the leading characters in the dictionary name are the same as all the characters in the data name found in the text. For example, the text `_atom_site.label_` would match the mmCIF dictionary entry **`_atom_site.label_alt_id`**.

(iii) If the data name is preceded by the asterisk character `*`, then search for a match in the dictionary where the trailing characters in the dictionary name are the same as all the characters in the data name found in the text. The first match found in the dictionary is accepted. For example, the text `*_alt_id` would match **`_atom_site.label_alt_id`**, or, if that name had not been in the dictionary, **`_struct_conn.ptnr1_label_alt_id`**. If one of the searches succeeds, add the line number of the data name to a list attached to the dictionary name. Up to 19 line numbers are retained for each dictionary name (the first ten matches and the last nine).

If no match is found, the unmatched data name is added to the list of unmatched names, along with the appropriate line number. If a data name has been misspelled it will be caught at this step.

When the text file has been processed, a validation report file is output containing the alphabetically sorted list of unmatched names and line numbers, followed by the sorted list of names from all dictionaries that are used within the text. If requested, this is followed by the sorted list of names from all dictionaries that are not used within the text in the file. If a data name has an alias defined in the dictionaries, a warning about the existence of the alias is given. If more than one dictionary has been used, the source dictionary is identified for each data name. An example of the output from *CYCLOPS* is shown in Fig. 5.3.4.1.

5.3.4.1.2. *Invocation of the program*

CYCLOPS is generally invoked from a command line that specifies the input and output file names and the dictionary files against which to validate the input. However, because the program is portable across a wide range of operating systems, there is substantial flexibility in the way in which it may be invoked. Under a Unix-like operating system, the program may typically be called with a command such as

`cyclops -i` *`infile`* `-o` *`outfile`* `-d` *`dictfile`*

where *infile* is the name of the input file for validation, *outfile* is the file to which the detailed output of the program is written and *dictfile* is a dictionary file.

A more complete set of options available in a Unix-like operating environment is

`cyclops` [`-i` *`infile`*] [`-o` *`outfile`*] [`-d` *`dictfile`*] [`-p` *`priority`*]
[`-f` *`cmndfile`*] [`-c` *`catck`*] [`-v` *`verbose`*] [`-s` *`short`*]

where the options are as follows:

-i specifies the name of the input file, *infile*.

-o specifies the name of the output file, *outfile*.

-d specifies the name of the dictionary file, *dictfile*. For compatibility with the original version of the software, the dictionary file may be *either* a CIF dictionary or a list of file names. That is, it may contain dictionary definitions in DDL format or (if the file begins with the characters `#DICT`) it may contain a list of dictionary file names to be entered. As implied by this last statement, multiple dictionaries may be specified to the program.

-p specifies the priority that should be assigned if multiple definitions for the same data name are encountered when multiple dictionaries are accessed. The permitted values are: *first* (the default), in which the first of duplicate definitions to be loaded takes priority; *final*, in which the last takes priority; and *nodup*, in which an instance of a duplicate definition should be treated as a fatal error.

-f specifies the name of a command file *cmndfile* that contains additional directives to the program.

-c is a flag indicating whether an error message should be raised if a data name has been assigned a category different from the leading portion of the data name itself. The Boolean variable *catck* may take the values 't', '1' or 'y' for *true*, 'f', '0' or 'n' for *false*.

-v is a flag indicating whether a verbose listing of unreferenced data names should be generated. The Boolean variable *verbose* may take the same values for *true* or *false* as above.

-s is a flag indicating whether the output should be short (*i.e.* restricted to items not in dictionaries). The Boolean variable *short* takes the same values as above.

For the flags expecting Boolean values, the default is 'f' (*false*).

If no input or output file names are specified, the program will read from the standard input channel or write to standard output,

```
           CYCLOPS Check List
           ------------------
   Dictionary data names  = 2244
   New data names in text =    4
    [1]  Dictionary cif_core.dic 2.0.1 data names =  624
    [2]  Dictionary cif_mm.dic 0.9.0 data names = 1620

Data names NOT in Dictionary           Line Numbers

_blat1 . . . . . .                     9   11   94  96
                                      181  183  290  296
_blat2 . . . . . .                    13   15   98  100
                                      185  187  287  293
_dummy_test  . . . . . .               5    7   90  92
                                      177  179  201
_rubbish_here. . . . . .              431

[1]  Dictionary cif_core_2.0.1.dic
[2]  Dictionary cif_mm.dic
                                      Line Numbers

[2] _atom_site.calc_attached_atom     413
[1] = _atom_site_calc_attached_atom   412
[2] _atom_site.calc_flag  . . . . .   410
[1] = _atom_site_calc_flag            409
[2] _atom_site.fract_x  . . . . . .    38   44   50  390
[1] = _atom_site_fract_x              389
[2] _atom_site.fract_y  . . . . . .    39   45   51  394
[1] = _atom_site_fract_y              393
[2] _atom_site.fract_z  . . . . . .    40   46   52  398
[1] = _atom_site_fract_z              397
[2] _atom_site.id   . . . . . .        37   43   49  386
[1] = _atom_site_label                385
[2] _atom_site.thermal_displace_type  406
[1] = _atom_site_thermal_displace_type 405
[2] _atom_site.type_symbol  . . . .   416  420  424  428
                                      434  438  442  450
[1] = _atom_site_type_symbol          415  419  423  427
                                      433  437  441  449
 [later in the validation output file, showing the transition to unreferenced data names . . . ]

[1] _symmetry_cell_setting  . . . .   319
[2] = _symmetry.cell_setting          320
[1] _symmetry_space_group_name_H-M    323
[2] = _symmetry.space_group_name_H-M  324
[1] _symmetry_space_group_name_Hall   327   445
[2] = _symmetry.space_group_name_Hall 328   446

[1]  Dictionary cif_core_2.0.1.dic
[2]  Dictionary cif_mm.dic
                                   Names Not Referenced

[2] _atom_site.aniso_B[1][1]
[2] _atom_site.aniso_B[1][1]_esd
[2] _atom_site.aniso_B[1][2]
 [. . . portion of output omitted . . .]

[2] _atom_site.aniso_U[3][3]_esd
[2] _atom_site.attached_hydrogens
[1] = _atom_site_attached_hydrogens
[2] _atom_site.auth_asym_id
[2] _atom_site.auth_atom_id
[2] _atom_site.auth_comp_id
[2] _atom_site.auth_seq_id
[2] _atom_site.B_equiv_geom_mean
[1] = _atom_site_B_equiv_geom_mean
[2] _atom_site.B_equiv_geom_mean_esd
[2] _atom_site.B_iso_or_equiv
[1] = _atom_site_B_iso_or_equiv
[2] _atom_site.B_iso_or_equiv_esd
 [. . . remainder of output omitted . . .]
```

Fig. 5.3.4.1. Sample output from *CYCLOPS*. The output has been edited and reformatted slightly to fit into the present column width.

respectively. The special character hyphen ('-') may also be supplied as an argument to '*-i*' or '*-o*' to indicate standard input or standard output.

Finally, if the operating system supports the passing of environment variables to a program, the names of the input file, output file and dictionary file may be passed through the values of `$CYCLOPS_INPUT_TEXT`, `$CYCLOPS_VALIDATION_OUT` or `$CYCLOPS_CHECK_DICTIONARY`, respectively.

5.3.5. File transformation software

This section describes a number of applications that transform an input CIF either to another CIF that contains a subset of the original contents or to other formats suitable for use with general processing tools. (Conversion to other crystallographic data formats is not discussed here.)

5.3.5.1. *QUASAR*: a data extractor

The oldest CIF manipulation program is *QUASAR* (Hall & Sievers, 1993), which was described as the prototype CIF application in the original standard specification paper (Hall *et al.*, 1991). Much of the functionality of *QUASAR* has now been included in the *cif2cif* program (Section 5.3.5.2). However, it remains useful as an application in its own right, and so is briefly described here.

5.3.5.1.1. *Purpose*

The program was designed to read a *request list* of data names, to locate the associated data in an input CIF and to output the data in the order of the request list. The output retains local conformance to CIF syntax rules, but the output file may not be strictly CIF conformant. For example, the same data can be requested multiple times and will be reproduced as often as requested in the output stream, a feature forbidden within a legal CIF.

5.3.5.1.2. *Mode of operation*

Written as a pure Fortran77 application, *QUASAR* requires three data streams: a file containing the request list, an input CIF and an output file. In an operating system such as Unix, it is convenient to attach the request list to the standard input channel; the first two lines of the input stream then take the form `star_arc_`*`infile`* and `star_out_`*`outfile`*, where *infile* and *outfile* are the file names of the input and output files, respectively.

The assignment of an output file may be replaced by a line containing `star_log`. When this is done, the program will test the syntactic validity of the input CIF and write any error messages to the standard output channel. In this mode the program may be used as a syntactic validator, although it is more tolerant of certain syntactic errors than *vcif* (Section 5.3.2.1).

5.3.5.1.3. *The request list*

Fig. 5.3.5.1 is an example request list, intended to highlight some of the special features of the way the program operates. Fig. 5.3.5.2 shows an example CIF against which this request list will be tested; Fig. 5.3.5.3 shows the output. Both figures have been modified slightly to fit on the printed page; they are derived from the sample files distributed with the program.

The request list begins with directives specifying the input and output file names (qtest.cif and qtest.out, respectively). The file may contain comments prefaced by a hash character `#`; this is a useful feature for annotating a request list. Another use for such comments is seen in the standard request list distributed to authors for papers published in *Acta Crystallographica*. Here, data names that are *not* normally published are hidden within the request list as comments and may be activated if they occur in a **`publ_manuscript_incl_extra_item`** loop within a CIF (see Section 5.7.2.3).

```
star_arc_qtest.cif
star_out_qtest.out

data_    #<< wild-card block name - accepts first

# request all fractional coord items
   _atom_site_fract_
   _atom_site_label
# capitals to test case insensitivity
   _atom_site_aniso_LABEL
# request something that is not in the CIF
   _dummy
   _atom_site_aniso_U_11

data_P6122
_        #<< this requests all data in this block
```

Fig. 5.3.5.1. An example request list for *QUASAR*.

```
data_P6122

loop_
_atom_type_symbol
_atom_type_oxidation_number
_atom_type_number_in_cell
   # capitals to test case insensitivity
_atom_type_scat_dispersion_REAL
_atom_type_scat_dispersion_imag
_atom_type_scat_source
  S   0  6   .319  .557
        Int_Tab_Vol_III_p202_Tab._3.3.1a
  O   0  6   .047  .032
        Cromer,D.T._&_Mann,J.B._1968_AC_A24,321.
  C   0  20  .017  .009
        Cromer,D.T._&_Mann,J.B._1968_AC_A24,321.
  RU  0  1  -.105   3.296
        Cromer,D.T._&_Mann,J.B._1968_AC_A24,321.

loop_
_atom_site_label
_atom_site_fract_x
_atom_site_fract_y
_atom_site_fract_z
_atom_site_U_iso_or_equiv
_atom_site_thermal_displace_type
_atom_site_calc_flag
_atom_site_calc_attached_atom
_atom_site_type_symbol
  s   .20200  .79800  .91667  .030(3)  Uij  ?  ?  s
  o   .49800  .49800  .66667  .02520  Uiso  ?  ?  o
  c1  .48800  .09600  .03800  .03170  Uiso  ?  ?  c

loop_
_atom_site_aniso_label
_atom_site_aniso_U_11
_atom_site_aniso_U_22
_atom_site_aniso_U_33
_atom_site_aniso_U_12
_atom_site_aniso_U_13
_atom_site_aniso_U_23
_atom_site_aniso_type_symbol
  s .035(4) .025(3) .025(3) .013(1) .000 .000 s
```

Fig. 5.3.5.2. Example CIF for demonstrating the use of *QUASAR*.

```
data_P6122

loop_
_atom_site_fract_x
_atom_site_fract_y
_atom_site_fract_z
_atom_site_label
  .20200 .79800 .91667 s
  .49800 .49800 .66667 o
  .48800 .09600 .03800 c1

loop_
_atom_site_aniso_label
_dummy           # requested item not present
_atom_site_aniso_U_11
  s ? .035(4)

#                -----end-of-data-block-----

data_P6122

loop_
_atom_type_symbol
_atom_type_oxidation_number
_atom_type_number_in_cell
_atom_type_scat_dispersion_REAL
_atom_type_scat_dispersion_imag
_atom_type_scat_source
  S  0  6  .319  .557
        Int_Tab_Vol_III_p202_Tab._3.3.1a
  O  0  6  .047  .032
        Cromer,D.T._&_Mann,J.B._1968_AC_A24,321.
  C  0 20  .017  .009
        Cromer,D.T._&_Mann,J.B._1968_AC_A24,321.
  RU 0  1 -.105 3.296
        Cromer,D.T._&_Mann,J.B._1968_AC_A24,321.

loop_
_atom_site_label
_atom_site_fract_x
_atom_site_fract_y
_atom_site_fract_z
_atom_site_U_iso_or_equiv
_atom_site_thermal_displace_type
_atom_site_calc_flag
_atom_site_calc_attached_atom
_atom_site_type_symbol
  s  .20200 .79800 .91667 .030(3) Uij  ? ? s
  o  .49800 .49800 .66667  .02520 Uiso ? ? o
  c1 .48800 .09600 .03800  .03170 Uiso ? ? c

loop_
_atom_site_aniso_label
_atom_site_aniso_U_11
_atom_site_aniso_U_22
_atom_site_aniso_U_33
_atom_site_aniso_U_12
_atom_site_aniso_U_13
_atom_site_aniso_U_23
_atom_site_aniso_type_symbol
  s .035(4) .025(3) .025(3) .013(1) .000 .000 s

#                -----end-of-data-block-----
```

Fig. 5.3.5.3. Result of running *QUASAR* with the example request list of Fig. 5.3.5.1 on the CIF listed in Fig. 5.3.5.2.

The request list must specify the data block from which the requested data are to be extracted. Multiple data blocks may be requested in the same file. An entry '`data_`' operates as a wild card and indicates that requests should be served from the next data block encountered. In the example above, the first group of requests will be met from the first data block in the CIF; the second set from the data block named 'P6122' (if present).

5.3.5.1.4. *Output from QUASAR*

The body of the request list is a series of data names. Where a data name appears in the CIF, it will be extracted with its associated data value or values. The user need not have prior knowledge of whether a data item occurs in a looped list or not: *QUASAR* will automatically retrieve the matching values and construct a loop header if necessary. However, because the requests are served in the exact order in which they occur in the file, data items in the same list in the input CIF may be extracted into different lists upon output. Although this breaks the semantic association between items grouped in the same list (especially for CIFs described by the DDL2 relational scheme), it is a syntactically valid construction and may be a valuable feature for some processes.

5.3.5.1.4.1. *Treatment of missing data*

When a requested data item is absent from the CIF, *QUASAR* will nevertheless emit a data name with a corresponding value of '?', the conventional CIF value of null type for 'unknown quantity'. A CIF comment is also generated by *QUASAR* to indicate that the entry was missing from the input CIF. If the missing data name is found between data names that have multiple values and that occur in the same looped list, it is assumed that the missing data name should be associated with the same looped list, and it will be emitted in the loop header; the integrity of the list is then satisfied by emitting a column of unknown values. Note how this behaviour differs from that of the generic STAR File extraction utility *Star_Base* (Spadaccini & Hall, 1994), which silently ignores missing data items. However, it is a useful behaviour for applications that depend on finding a specific data item in their processing stream, even where its value is unknown.

5.3.5.1.4.2. *Matching data names*

As with the specification of data-block names, the data names in the request list may have a trailing underscore. Where this is the case, *QUASAR* will retrieve all data items where the data name starts with the specified string. For example, a request for '`_atom_site_`' will extract *all* data names starting with '`_atom_site_`'. The special case of an isolated underscore character '_' matches *all* data names present in the current data block.

5.3.5.1.4.3. *Case sensitivity*

The example demonstrates the way in which the application handles the case insensitivity of a requested data item. Data names are converted internally to a lower-case representation, both from the request list and the input CIF. Matches are therefore determined in a case-insensitive manner. However, if a data name is present in the CIF, its original case is retained on output. This permits the computationally irrelevant but cosmetically useful retention of capitalization as used in canonical CIF dictionary definitions. Where the requested data name is absent, the output is all lower-case.

5.3.5.2. *cif2cif*

cif2cif (Bernstein, 1998) is a program built with the *CIFtbx* toolkit (Chapter 5.4) to copy a CIF while checking data names against dictionaries, optionally reformatting numbers to maintain standard uncertainties within a specified range. The output CIF may contain a subset of the data in the original CIF according to a request list, in the manner of *QUASAR* (Hall & Sievers, 1993).

The program was built as a sample application using *CIFtbx* routines and grew out of requirements from several sources.

5.3.5.2.1. *Operation*

5.3.5.2.1.1. *Copying*

In its simplest application, the program copies a CIF from the standard input channel to standard output. The copy is not verbatim (standard utilities of the computer operating system should be used for that purpose), but the output CIF differs from the input only in the following respects: some comments are deleted; lines in the input longer than 80 characters are wrapped to 80 characters or less; white space between tokens may be altered, especially in an attempt to align entries in looped lists in a cosmetically pleasing manner. While none of these changes should affect robust CIF-parsing applications, they are nevertheless useful in imposing a uniform style of presentation for browsing in a text editor or other human-readable framework.

5.3.5.2.1.2. *Constraining standard uncertainties to specified ranges*

Some journals require that standard uncertainties in experimental values should be quoted within a specified range. Typically the standard uncertainty (s.u.) should be quoted as an integer in parentheses, modifying the last place or two of decimals in the experimental data, and with a value between 2 and 19. *cif2cif* permits s.u. values in the ranges 1–9, 2–19 or 3–29, selectable by a command-line switch. The effect of applying the 'rule of 19' would be to change a value of 1.458(1) in the input CIF to 1.4580(10) in the output.

5.3.5.2.1.3. *Dictionary validation*

cif2cif will open one or more CIF dictionary files as it copies the input CIF and identify certain classes of error against the dictionary definitions. The conditions that will raise an error are an unrecognized data name or a wrong data type. The program will also optionally indicate a warning if a data name has been assigned a category different from the leading portion of the data name – this may indicate an inconsistency within the dictionary itself.

5.3.5.2.1.4. *Serving a request list*

cif2cif will extract a subset of the data items contained in a CIF as specified by a request list, in the manner of *QUASAR*. The handling of data names specified in the request list is as described in Section 5.3.5.1.3 above, with the following additional feature. The special string `data_which_contains:` will extract the specified data items from the first data block in which at least one occurs; the block code need not be known in advance.

Some care must be exercised in attempting to extract data from data blocks by context without prior knowledge of the file contents. Consider the following simple example file:

```
data_A
   loop_    _A1
            _A2
             a1 a2 aa1 aa2


data_B
   loop_    _A1
            _B1
             a b aa bb
```

The loop containing **`_A1`** and **`_B1`** *cannot* be extracted with a request list of the form

```
data_which_contains:
_A1
_B1
```

because **_A1** occurs in the first data block encountered; the output from *cif2cif* in this example will be

```
data_A
    loop_     _A1
               a1   aa1
#             ---end-of-data-block---
```

The behaviour of the program differs from *QUASAR* in two other small ways. When the request list forces the output data stream to contain the same data-block header more than once, an error message is posted to the standard error channel and the data-block headers in the output stream are annotated with a comment of the form '`#<---- duplicate data block`'. In this case the output file does *not* conform to the CIF syntax rules.

When a data name is requested but no matching data item appears in the output file, *cif2cif* writes an error message to the standard error channel. However, unlike *QUASAR*, which inserts the requested data name in the output stream with an associated value of '`?`' (for *unknown*), *cif2cif* produces *no* output for the requested data item.

5.3.5.2.1.5. *Other features*

Some additional features are of use in special circumstances.

The user may preserve the layout of the contents of looped lists exactly as in the input file, or may ask the program to adjust the layout to a more visually pleasing tabular form.

The user may enable recognition of data-name aliases in the dictionaries used for validation. When the relevant command-line argument is set to *true*, user-supplied data names will be transformed to the canonical forms in the validating dictionary. This would permit, for example, a small-molecule CIF using the core dictionary definitions to be converted to mmCIF format.

The user may prefix each line of output with an identical character string. A typical reason for so doing would be to include a fragment of CIF listing within the body of an email message or some other document. Such an output would not conform to the syntax rules for CIF.

5.3.5.2.2. *Invocation of the program*

cif2cif is another application of the *CIFtbx* library by the same author, and so has a similar user interface to that of *CYCLOPS* (Section 5.3.4.1.2). Under a Unix-like operating system, the program is typically called with a command such as

`cif2cif -i` *infile* `-o` *outfile* [`-q` *reqfile*]

where *infile* is the name of the input file, *outfile* is the output file and *reqfile* is an optional file containing a request list for a subset of the original contents.

A more complete set of options available in a Unix-like operating environment is

`cif2cif` [`-i` *infile*] [`-o` *outfile*] [`-d` *dictfile*] [`-q` *reqfile*] [`-f` *cmndfile*] [`-c` *catck*] [`-a` *alias*] [`-t` *tab*] [`-e` *sulim*] [`-p` *prefix*]

where the options are as follows:

-i specifies the name of the input file, *infile*.

-o specifies the name of the output file, *outfile*.

-d specifies the name of a dictionary file, *dictfile*, against which the existence, type and category of data names are checked. The dictionary file may be *either* a CIF dictionary or a list of file names. That is, it may contain dictionary definitions in DDL format or (if the file begins with the characters **#DICT**) it may contain a list of dictionary file names to be entered. Thus, multiple dictionaries may be specified to the program.

-q specifies the name of the request file, *reqfile*, containing a list of data names (with associated data-block directives) that should be extracted as a subset of the contents of the original file.

-f specifies the name of a command file *cmndfile* that contains additional directives to the program.

-c is a flag indicating whether an error message should be raised if a data name has been assigned a category different from the leading portion of the data name itself. The Boolean variable *catck* may take the values 't', '1' or 'y' for *true*, 'f', '0' or 'n' for *false*.

-a is a flag indicating whether data-name aliases in the validating dictionary should be used to replace user-supplied names by their canonical forms. The Boolean variable *alias* may take the same values for *true* or *false* as above.

-t is a flag indicating whether the output should be reformatted with tabs to produce a regular table layout within looped lists. The Boolean variable *tab* takes the same values as above. If *true*, text is reformatted; if *false*, the original formatting is retained.

For the flags expecting Boolean values, the default is 'f' (*false*).

-e specifies the precision to retain in rounding standard uncertainty values. The permitted integer values are 9, 19 (the default) and 29.

-p takes a string value which is prefixed to every line of output. Every occurrence of the underscore character '_' in the prefix is changed to a space on output.

If no input or output file names are specified, the program will read from the standard input channel or write to standard output, respectively. The special character hyphen ('`-`') may also be supplied as an argument in place of a file name to indicate standard input or standard output as appropriate.

Finally, if the operating system supports the passing of environment variables to a program, the name of the input file may be passed as the value of `$cif2cif_INPUT_CIF`, and likewise the output file, `$cif2cif_OUTPUT_CIF`, dictionary file, `$cif2cif_CHECK_DICTIONARY`, and request file, `$cif2cif_REQUEST_LIST`, may be specified.

5.3.5.3. *ciftex*: translating to a typesetting language

The program *ciftex* (McMahon, 1993) was developed to create files for typesetting the journal *Acta Crystallographica* using the text-formatting language TEX (Knuth, 1986). Details of its use in the journal production process are given in Chapter 5.7. It is discussed here as an example of translating a CIF to some output format where data values are annotated with different text depending on their accompanying data names.

5.3.5.3.1. *Basic operation of ciftex*

The program is designed to act as a filter, typically in a Unix-style environment, reading a CIF on the standard input channel and outputting a modified data stream to standard output. The output is a file of TEX code that is processed by the TEX program to produce a device-independent file describing the content of a formatted typeset document. Further post-processing allows the formatted document to be viewed on the screen or printed.

Each input token (number, character or text string; data name; **loop_** or **data_** keywords) is transformed as it is identified; there is no lookahead and minimal retention of context. The data stream is treated purely syntactically; no transformations are applied on the basis of the supposed meaning of any of the file contents.

```
_cell_formula_units_Z        2
_cell_length_a               8.79(2)
_refine_ls_extinction_coef    .347e4(5)
_chemical_name_common        'copper sulphate'
```

Fig. 5.3.5.4. Sample CIF data input to *ciftex*.

```
\cellz{2}
\nobreak\cella{8.79 (2)}
\extcoeffLarson{0.347 (5) $\times$ $10^{4}$}
\chemcom{copper sulfate}
```

Fig. 5.3.5.5. Output from *ciftex* run on the data of Fig. 5.3.5.4.

5.3.5.3.1.1. *Non-looped data*

For portions of the CIF that are not contained in looped lists, the transformations are trivial. A (*data name*, *data value*) pair is transformed to a TEX macro and its argument. The macro name is determined from an external 'map' file which the program reads at run time; this file associates CIF data names and the corresponding TEX macros through a simple lookup table.

A CIF data value is in most cases passed as the argument to the corresponding TEX macro with few modifications. If the data value is a character string beginning with an integer, full point, hyphen or plus character, it is assumed to be of type 'numb'. A space is introduced ahead of an embedded open parenthesis (to separate a standard uncertainty from its parent value). A leading zero is printed before any bare decimal point. An embedded E is taken to indicate exponential notation and the format of the number is accordingly modified.

If the input data value is of type 'char' (*i.e.* is a single token beginning with characters other than those recognized as the leading characters for numerical data; or contains multiple tokens delimited by quote marks or semicolons), the program will search the map file for key values exactly matching each token, and if found will substitute the token by its replacement word or text. If no replacement is specified in the map file, the token is passed unchanged to the standard output channel. This facility was found to be useful in making global substitutions of individual words during file processing, but must be used with care since the substitutions are unconditional, without any reference to context.

Some small examples of typical non-looped data items are shown in Fig. 5.3.5.4 and the corresponding *ciftex* translation based on a map file used for typesetting *Acta Crystallographica Section C* is shown in Fig. 5.3.5.5.

Note the transformations of the numerical arguments and the translation of 'sulphate' to 'sulfate'.

5.3.5.3.1.2. *Looped data*

If the input token is a `loop_` keyword, the program enters a different mode of operation. Looped data may be represented in print either as repetitive lists or in tabular format. There is no indication in a CIF dictionary of the appropriate representation (nor should there be, for what is essentially a matter of presentation) and the choice is made based on a flag associated with each data name in the map file. For non-tabular lists, the structure

```
loop_
    _dataname_1
    _dataname_2
       value_1    value_2
       value_3    value_4
```

```
\settabs 5 \columns
\+\relax & $x$  &   $y$    &  $z$   &$U_{\rm eq}$&\cr
\+Re &0.222 (1) &0.003 (1) &0.146 (1) &0.042 (1) &\cr
\+Co &0.234 (1) &0.139 (1) &0.299 (1) &0.046 (1) &\cr
\+P1 &0.358 (1) &0.222 (1) &0.197 (1) &0.044 (1) &\cr
\+P2 &0.106 (2) &0.051 (1) &0.289 (1) &0.046 (1) &\cr
\+C1 &0.308 (6) &0.029 (6) &0.034 (4) &0.057 (4) &\cr
\+O1 &0.356 (5) &0.044 (5) &0.030 (3) &0.079 (3) &\cr
\+C2 &0.066 (6) &0.039 (6) &0.111 (4) &0.056 (4) &\cr
```

Fig. 5.3.5.6. TEX markup for typesetting a table of atomic coordinates.

is translated to a sequence of TEX codes of the form

```
\macro_one(value_1)
\macro_two(value_2)
\macro_one(value_3)
\macro_two(value_4)
```

In the case of tabulated data, the `loop_` header is translated into a set of table headings and typographic codes are introduced to lay out in columnar format the values in the body of the list. The number of different data names in the loop header is counted and the data values are identified by their position in the loop modulo the total number of data names in the header (in effect, by their 'phase' in the loop). In the simplest case, a TEX command is emitted that builds a table with *n* columns, where *n* is the number of different data names. Then the data values are counted as they are processed. After every *n*th data value, a TEX code is emitted indicating 'end of table row' and a further code is emitted before the next value (if there is one) that means 'beginning of new table row'. In all other cases, a code is emitted signifying 'move to next column'.

Fig. 5.3.5.6 is a simplified extract from a table of atomic coordinates derived from the `_atom_site_` loop in a CIF.

5.3.5.3.1.3. *The ancillary map file*

The translation between a CIF data name and its replacement text in the TEX output file is defined in the external map file. The format of the translation is very simple, as illustrated in Fig. 5.3.5.7.

Each line starts with a CIF data name, which is terminated by a space character. The next character is either 'T' or 'N' to indicate whether the output should be tabulated or not. The next character is an arbitrary character from the ASCII character set, and is chosen to collect together data that will appear in the same logical section of the output file. This locator character may be associated, in another ancillary file described below, with additional text for output. The remainder of the line is the replacement text.

In the example supplied, the cell-length parameters map to the TEX macros `\cella`, `\cellb` and `\cellc` (each preceded by a standard TEX macro forbidding a page break immediately before the contents are printed). The details of the publication authors are described by a set of TEX macros that will occur in two different locations in the output file (the authors' names and addresses may be looped together in the location labelled by the character `a`; any explanatory footnotes and email addresses will be printed elsewhere in the paper, at the location labelled `x`). The anisotropic displacement parameters U^{ij} will be printed in a table and the replacement text consists of the TEX codes that will be printed at the head of each column in the table.

The initial text on the line need not be a CIF data name; it may be any other single word. In this case, every occurrence of that word in the input CIF will be replaced by the replacement text.

```
_cell_length_a Ng\nobreak\cella
_cell_length_b Ng\nobreak\cellb
_cell_length_c Ng\nobreak\cellc

_publ_author_name Na\author
_publ_author_address Na\address
_publ_author_footnote NX\aufootnote
_publ_contact_author_email NX\email

_atom_site_aniso_label TU\relax
_atom_site_aniso_U_11 TU{\hfill $U^{11}$ \hfill}
_atom_site_aniso_U_12 TU{\hfill $U^{12}$ \hfill}
_atom_site_aniso_U_13 TU{\hfill $U^{13}$ \hfill}
_atom_site_aniso_U_22 TU{\hfill $U^{22}$ \hfill}
_atom_site_aniso_U_23 TU{\hfill $U^{23}$ \hfill}
_atom_site_aniso_U_33 TU{\hfill $U^{33}$ \hfill}
```

Fig. 5.3.5.7. Example map file for use with *ciftex*.

```
#[:\newif\ifproof \prooftrue
#]:\iftwocol\vfill\enddoublecolumns\fi
#a:\pretolerance1000\parskip0pt\tolerance5000
#a:\vskip10pt
#g:
#g:%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#g:\iftwocol\enddoublecolumns\twocolfalse\fi
#g:\tenbf Experimental
#g:\noindent\ninebf Compound \datablock\vskip2pt
#g:\noindent\nineit Crystal data\par
#g:\vskip2pt\begindoublecolumns\twocoltrue\defaultfont
#U:%%%%%%%%%%% Table of anisotropic U's %%%%%%
#U:\iftwocol\enddoublecolumns\twocolfalse\fi
#U:\rm Table \tableno. \it Anisotropic displacement
#U:parameters \rm (\AA$^2$) for \datablock
#U:\vskip 6pt
```

Fig. 5.3.5.8. Example format file for *ciftex*.

If the initial *character* of the line is a hash mark `#`, the line is treated as a comment and discarded.

5.3.5.3.1.4. *The ancillary format file*

Because a printed paper may be more verbose than its parent CIF data file, it is necessary to add text to the output from *ciftex* to represent section headings, line spaces or other formatting instructions. The program reads an ancillary file, known as the format file, for such additional text.

Each line in the format file begins with a hash mark `#`, a single ASCII character and a colon. The second character is chosen to match the corresponding locator character associated with data names in the map file. The rest of the line is text to be output. When the locator character associated with the data name currently being processed differs from the previous one, the output text from all lines in the format file with the new locator character are output.

The special strings `#[:` and `#]:` indicate text to be emitted at the beginning and end of the output stream, respectively.

Fig. 5.3.5.8 is an example of a simplified format file. The first line is printed at the start of the output TEX file; the second line at the end. The next line will be printed on the first occurrence of a data name flagged with the locator code `a` in the map file. In this example, that will be the name or address of an author of the paper; some typographic directives are emitted immediately before the authors' names and addresses, including the introduction of a blank line ('vertical skip', or 'vskip') of height 10 typographic points.

The lines beginning `#g:` are emitted immediately before the first data name in the group that is associated with locator code `g`. In this example, the effect is to output a heading and subheading before printing the cell-length parameters and to switch to double-column format. The line containing *only* the characters `#g:` provides for the introduction of a blank line into the TEX file, with the sole purpose of making the file more readable by human editors.

The lines beginning `#U:` are emitted at the beginning of the table of anisotropic U values.

The mechanism looks complicated at first sight, but addresses the need to generate headings at standard locations in a printed paper when the exact content of the paper is not known in advance.

The different format for directives in the map and format files means that the same file can be used for both purposes, if required. In practice it is often easier to maintain different files: the same mapping between CIF data names and TEX macros might be common to different journals, while each journal uses its own format file.

5.3.5.3.2. *Invocation of the program*

The program reads a CIF on the standard input channel and outputs TEX code on standard output. There is no provision to specify file names. It is therefore invoked within a Unix-style operating system by a command such as

`ciftex` < *infile* > *outfile*

where *infile* and *outfile* are the input and output files respectively; or it may be called as part of a pipeline of procedures:

program 1 < *infile* | `ciftex` | *program 2* ...

A number of command-line options may be supplied to modify the operation of the program. Other than the specification of the map and format files, they are largely relevant to differing house styles for IUCr journals.

The options -map *mapfile* and -format *formatfile* specify the names of the ancillary map and format files. If not specified, they are sought in default locations on the user's file system (different values may be defined when the program is compiled) or as specified in the environment variables `$CIFTEX_MAP` and `$CIFTEX_FORMAT`, respectively.

The options *-H* and *-N* specify, respectively, whether or not hydrogen atoms in coordinate tables should be printed. The hydrogen-atom lines in the table are in fact always emitted on standard output, but in the case of the *-N* option are prefixed by a `%` (TEX comment) character and so ignored by TEX.

Options *-c* and *-F* specify the printing of centred decimal points or commas for decimal points, respectively. Finally, the option *-d* modifies certain assumptions that *ciftex* makes when typesetting CIF dictionaries. The details are of interest only to a specialist.

5.3.5.3.3. *Some general comments*

Although *ciftex* is available for public use and redistribution within the academic community, it is clearly of most interest to users who need to generate typeset representations of the contents of CIFs. Nevertheless, some elements of its design are relevant to other applications that perform on-the-fly file transformations on a strictly syntactic basis.

First, the functionality is very simple, essentially tokenizing the input data stream and exchanging tokens for replacement text as directed. An immediate consequence of this is the need for additional utilities to manipulate the input file if, for example, the data need to be presented in a particular order. In the journals production process, *QUASAR* is used to reorder an input file before passing it to *ciftex*.

Second, the replacement text should be externalized as much as possible. The use of map and format files means that the same basic program can be used for formatting according to any set of typographic rules; only the ancillary files need to be modified. In the current version of *ciftex*, the program performs some replacements internally; an objective of further development is to remove this function from the program and to externalize it either in more sophisticated table lookup files or in separate methods modules.

Third, the concept of replacement should be abstracted as much as possible. The software was written initially with the objective of replacing data names with TEX macros. Experience suggests that a generic transformation program could be written with the philosophy of replacing data names and data values by directives implemented before and after the occurrence of the data value, and as events upon its first, last and intervening occurrences. Such 'directives' and 'events' could be mapped to arbitrary replacement strings in any markup scheme, such as SGML, XML, HTML, TEX, LATEX or commercial word-processing encodings.

5.3.6. Libraries for scripting languages

Recent years have seen a great increase in the popularity of scripting languages – broadly, computer languages where the source code is run-time interpreted and that have powerful facilities for interacting with other processes and operating-system utilities. In part this is because such languages lend themselves to rapid prototyping; but the powerful features of languages such as Perl (Wall *et al.*, 2000), Python (van Rossum, 1991) and Tcl/Tk (Ousterhout, 1994) make it entirely feasible to create and maintain complex programs that can run efficiently. Several of the applications discussed in this chapter make effective use of one or more of these languages, either solely or alongside more traditional compiled languages.

As the scripting languages have become more powerful, they have also acquired more structure, so that authors now frequently build libraries (or 'modules') of functions or subroutines that can be re-used in a range of applications. This is a welcome development, for the availability of public libraries not only reduces the effort required to develop new applications, but also goes a long way towards establishing common application programming interfaces that help to standardize the way in which software from different sources is developed.

In this section two available libraries of this type are reviewed: *STAR::Parser* and *PyCifRW*.

5.3.6.1. *STAR::Parser* and related Perl modules

A collection of Perl modules has been developed at the San Diego Supercomputer Center (Bluhm, 2000) to provide basic library routines for object-oriented manipulation of STAR files with restricted syntax appropriate to CIF applications. The module name suggests that a more complete STAR implementation may be considered in future developments, but at present the modules do not handle nested loops or the inheritance of data values from global blocks. Indeed, they are still rather limited in scope; nevertheless, for the programmer wishing to prototype CIF applications in Perl, they offer a very rapid entry to parsing CIFs and constructing useful data structures that can be manipulated with standard Perl tools.

The use of some of the modules is illustrated in Fig. 5.3.6.1, which is a simplified version of the main program loop in the application written to typeset the CIF dictionaries printed in Part 4 of this volume.

5.3.6.1.1. *STAR::Parser*

STAR::Parser is the basic parsing module and may parse either data files or dictionaries (which may include save frames). It contains a single class method, *parse*, which returns an array of DataBlock objects. Each DataBlock object contains all the data items within an individual data block of the file. Even if the file contains only a single data block, the resulting object is passed in an array.

The contents of the data blocks may be accessed and manipulated by the methods provided by the *STAR::DataBlock* and *STAR::Dictionary* modules. They are stored internally as a multi-dimensional hash (the Perl term for an associative array with keys and associated values, which may themselves be complex data objects). Keys are provided for data blocks, save blocks, categories and data items identified during the parse. The module provides no error checking of files or objects, against the dictionary or otherwise – limited checking functionality is available through other modules in this collection.

In the example of Fig. 5.3.6.1, the *parse* method is called at line 9 to read a DDL2 dictionary file (indicated by the `-dict=>1` parameter) and return an array of data blocks. In DDL2 dictionaries (such as the mmCIF dictionary of Chapter 4.5) an entire dictionary is contained within a data block; save frames partition the data block into definitions for separate items. Normally a DDL2 CIF dictionary has only a single data block; nevertheless, the example program can handle multiple data blocks in the array, and traverses the one or several data blocks in the array through a Perl *foreach* construct (line 15).

5.3.6.1.2. *STAR::Dictionary*

The *STAR::Dictionary* module contains class and object methods for Dictionary objects created by the *STAR::Parser* module, and is in fact a subclass of *STAR::DataBlock* (see next section). Since CIF dictionaries are fully compliant STAR files, they require little that is different from the methods developed for handling data files. The method *get_save_blocks* is provided to return an array of all save frames found in the Dictionary object.

In line 25 of the example, the method is called on each dictionary loaded from the input file (as described above, normally there will only be one). The method is combined with the Perl *sort* function to create an array of save frames from the dictionary, arranged in alphabetic order. All further manipulations of the contents of these save frames will use the methods of the generic *STAR::DataBlock* class.

5.3.6.1.3. *STAR::DataBlock*

This package provides several useful methods for handling the objects within a data block returned by the *STAR::Parser* module.

The class has a constructor method *new*, which can create a completely new DataBlock object if called with no argument. This is of course essential for applications that wish to write new CIFs. Alternatively, it may be called with a `$file` argument to retrieve an existing object that has previously been written to the file system using the *store* object method described below:

```
$data_obj = STAR::DataBlock->new{ -file=>$file };
```

Table 5.3.6.1 summarizes the object methods provided by the package. The *store* method allows a DataBlock object to be serialized and written to hard disk for long-term storage. The Perl public *Storable::* module is used.

The *get_item_data* method returns the data values for a named data item. It is used frequently in the example program of Fig. 5.3.6.1; for example, at lines 28–30 the array of categories

```
 1 #!/usr/local/bin/perl
 2
 3 use STAR::Parser;
 4 use STAR::DataBlock;
 5 use STAR::Dictionary;
 6
 7 my $file = $ARGV[0];
 8
 9 @dicts = STAR::Parser->parse(-file=>$file, -dict=>1);
10
11 exit unless ( $#dicts >= 0 ); # exit if nothing there
12
13 print_document_header;
14
15 foreach $dictionary (@dicts) { # usually one
16   # Get the dictionary name and version
17   $dictname = ($dictionary->get_item_data(
18                            -item=>"_dictionary.title"))[0];
19   $dictversion = ($dictionary->get_item_data(
20                          -item=>"_dictionary.version"))[0];
21
22   print "Dictionary $dictname, version $dictversion\n";
23
24   # Get list of save-frame names, sorted alphabetically
25   @saveblocks = sort $dictionary->get_save_blocks;
26
27   foreach $saveblock (@saveblocks) { # For each save frame
28     @categories = $dictionary->get_item_data(
29                                    -save=>$saveblock,
30                                    -item=>"_category.id");
31     if ( $#categories == 0 ) { # category save frame
32       format_category($saveblock); # format the category
33
34       $category = $categories[0] ;
35       foreach $block (@saveblocks) { # re-traverse blocks
36         @itemcategoryids = $dictionary->get_item_data(
37                                    -save=>$block,
38                                -item=>"_item.category_id");
39         @itemname = $dictionary->get_item_data(
40                                    -save=>$block,
41                                    -item=>"_item.name");
42
43         if ( $#itemcategoryids == 0) { # category pointer
44           format_item($block)
45                if $itemcategoryids[0] =  /^$category$/i;
46         } elsif ( $#itemname == 0 &&
47                     $itemname[0] =  /^_$category\./i ) {
48           format_item($block);
49         }
50
51       } # end foreach of items matching current category
52     }
53   } # end foreach of save frames in the dictionary
54 } # end foreach loop per included dictionary
55
56 print_document_footer;
57 exit;
```

Fig. 5.3.6.1. Skeleton version of an application to format a CIF dictionary for publication. Only the main program fragment is shown. Line numbering is provided for referencing in the text. *format_category*, *format_item* and the *print_document_** commands are calls to external subroutines not included in this extract.

in the input dictionary is assembled by retrieving the value associated with the data item **_category.id** as the component save frames are scanned in sequence. Note that for applications within dictionaries, the method takes a `-save=>$saveblock` parameter to allow the extraction of items from specific save frames. For manipulations of data files, this parameter is omitted. In lines 17–20, the method is called without this parameter because the name and version of the dictionary are expected to be found in the outer part of the file, not within any save frame.

get_keys allows a user to display the structure of a CIF or dictionary file and can be used to analyse the content of an unknown input file. When written to a terminal, the string that is returned by this method appears as a tabulation of the items present at the different levels in the data structure hierarchy, each level in the hierarchy being indicated by the amount of indentation (Fig. 5.3.6.2).

Table 5.3.6.1. *Object methods provided by the STAR::DataBlock Perl module*

Method	Description
store	Saves a DataBlock object to disk
get_item_data	Returns all the data for a specified item
get_keys	Returns a string with a hierarchically formatted list of hash keys (data blocks, save blocks, categories and items) found in the data structure of the DataBlock object
get_items	Returns an array with all the items present in the DataBlock
get_categories	Returns an array with all the categories present in the DataBlock
insert_category	Inserts a category into a data structure
insert_item	Insert an item into a data structure
set_item_data	Sets the data content of an item according to a supplied array

The *get_items* and *get_categories* methods are largely self-explanatory. The items or categories in the currently active DataBlock object are returned in array context.

insert_item and *insert_category* are the complements of these methods, designed to allow the insertion of new items or categories. Where appropriate (*i.e.* in dictionary applications), the save frame into which the insertion is to be made can be specified.

The remaining method, *set_item_data*, is called to set the data of item `$item` to an array of data referenced by `$dataref`:

```
$data_obj->set_item_data( -item=>$item,
                          -dataref=>$dataref );
```

As usual, an optional parameter `-save=>$save` may be included for dictionary applications where a save frame needs to be identified; the value of the variable *$save* is the save-frame name.

Note that the current version of the module does not support the creation and manipulation of data loops, although the *get_item_data* method will correctly retrieve arrays of data values from a looped list.

There are five methods available to set or retrieve attributes of a DataBlock object, namely: *file_name* for the name of the file in which the DataBlock object was found; *title* for the title of the DataBlock object (*i.e.* the name of the CIF data block with the leading **data_** string omitted); *type* for the type of data contained – 'data' for a DataBlock object but 'dictionary' for an object in the STAR::Dictionary subclass; and *starting_line* and *ending_line* for the start and end line numbers in the file where the data block is located. The method *get_attributes* returns a string containing a descriptive list of attributes of the DataBlock object.

5.3.6.1.4. *STAR::Checker*

This module implements a set of checks on a data block against a dictionary object and returns a value of '1' if the check was successful, '0' otherwise. The check tests a specific set of criteria:

(i) Are all items in the DataBlock object defined in the dictionary?

(ii) Are mandatory items present in the data block?

(iii) Are dependent items present in the data block?

(iv) Are parent items present?

(v) Do the item values conform to item type definitions in the dictionary?

Obviously, these criteria will not be appropriate for all purposes, and are in any case fully developed only for DDL2 dictionaries. An optional parameter `-options=>'1'` may be set to write a list of specific problems to the standard error output channel.

```
 data  save
block  block  categ. item
---------------------------

cif_img.dic
        -
                _category_group_list
                        _category_group_list.description
                        _category_group_list.id
                        _category_group_list.parent_id
                _dictionary
                        _dictionary.datablock_id
                        _dictionary.title
                        _dictionary.version
                _dictionary_history
                        _dictionary_history.revision
                        _dictionary_history.update
                        _dictionary_history.version
                _item_type_list
                        _item_type_list.code
                        _item_type_list.construct
                        _item_type_list.detail
                        _item_type_list.primitive_code
                _item_units_conversion
                        _item_units_conversion.factor
                        _item_units_conversion.from_code
                        _item_units_conversion.operator
                        _item_units_conversion.to_code
                _item_units_list
                        _item_units_list.code
                        _item_units_list.detail
        ARRAY_DATA
                _category
                        _category.description
                        _category.id
                        _category.mandatory_code
                _category_examples
                        _category_examples.case
                        _category_examples.detail
                _category_group
                        _category_group.id
                _category_key
                        _category_key.name
```

Fig. 5.3.6.2. Structure of the imgCIF dictionary (Chapter 4.6) as described by the *get_keys* method of the *STAR::DataBlock* module. Only the high-order file structure and the contents of the first category are included in this extract.

5.3.6.1.5. *STAR::Writer and STAR::Filter*

Two other modules are supplied by this package. *STAR::Writer* is a prototype module that can write STAR::DataBlock objects out as files in different formats; currently only the *write_cif* method exists to output a conformant CIF. *STAR::Filter* is an interactive module that prompts the user to select or reject individual categories from a STAR::Dictionary object when building a subset of the larger dictionary.

5.3.6.2. *PyCifRW*: CIF reading and writing in Python

PyCifRW (Hester, 2006) is a simple CIF input/output utility written in Python. It does not validate content against dictionaries, but it does provide a robust parser that has been extensively tested against various test files containing subtle syntactic features and against the collection of over 18 000 macromolecular CIFs available from the Protein Data Bank. The parser was implemented using the *Yapps2* parser generator (Patel, 2002) and is based on the draft Backus–Naur form (BNF) developed during a community exercise to review the CIF specification of Chapter 2.2.

As with the Perl library discussed above, *PyCifRW* presents an object-oriented set of classes and methods. Two classes are provided, *CifFile* and *CifBlock*.

A *CifFile* object provides an associative array of *CifBlock* objects, accessed by data-block name.

The methods available for the *CifFile* type are: *ReadCif(filename)*, which initializes or reinitializes a file to contain the CIF contents; *GetBlocks()*, which returns a list of the data-block names in the file; *NewBlock(blockname, [block contents])*, which adds a new data block to the file object; and *WriteOut(comment)*, which returns the contents of the current file as a CIF-conformant string, with an optional comment at the beginning. For the *NewBlock* method, the optional *block contents* must be a valid *CifBlock* object, as described below. The *NewBlock* method returns the name of the new block created, which will *not* be the requested *blockname* if a data block of the same name already exists (this conforms to the STAR and CIF requirement that data-block names must be unique within a file).

A *CifBlock* object represents the contents of a data block. The methods available to retrieve or manipulate the contents are: *GetCifItem(itemname)*, which will return the value of the data item with data name given by *itemname* (and which can be a single value or an array of looped values); *AddCifItem(data)*, which adds *data* to the current block, where *data* represents either a data name and an associated single value, or, for the case of looped data, a tuple containing an array of data names and an array of arrays of associated data values; *RemoveCifItem(dataname)*, to remove the specified data item from the current block; and *GetLoop(dataname)*, which returns a list of all data items occurring in the loop containing the data name provided. If *dataname* does not represent a looped data item, an error is returned.

The *GetLoop* method is important for the proper handling of looped data, and care is taken to handle loops robustly and efficiently. Items that are initially looped together are kept in the same loop structure.

Both *CifFile* and *CifBlock* objects act as Python mapping objects, which has the advantage that the value of a data item can be read or changed using an intuitive square-bracket notation. For example, a program can retrieve the value of a data item named `_my_data_item_name` in block ‘myblockname’ of a previously opened file *cf* using the following syntax:

```
value = cf["myblockname"]["_my_data_item_name"]
```

The returned value is either a single item, or a Python array of values if the data item occurs in a loop. Values are set in an analogous way and other common operations with mapping objects are also implemented.

5.3.7. Rapid development tools

The programs described so far in this chapter tend to fulfil a single purpose. Each program addresses a single clearly defined task and is of benefit to a user who has no need or desire to write a customized program. Where there is a need to write a new application, libraries of subroutines are available to the full-time programmer, such as those described in Chapters 5.4 to 5.6. However, in between these extremes, there are a large number of cases for which there is a need to combine the functionality of a number of existing programs without incurring the overhead of writing a new integrated application.

This section describes utilities that assist the development of new applications from existing software.

5.3.7.1. ZINC: an interface to CIF for standard Unix tools

Unix and derivative operating systems provide a convenient environment for rapid prototyping of programs acting on textual data. The convenience arises from the facility to chain applications together in a 'pipeline', where the output from an application may be passed directly to the input channel of another application without needing intermediate disk files for storage; and from the very rich set of utilities supplied with the typical Unix command shell, which permit files to be concatenated, split, compared, searched, stream-edited and otherwise transformed.

Many users are familiar with these utilities and can rapidly develop prototype or short-lived applications of great power by chaining them together as required. There is a temptation to use such techniques to manipulate CIFs, which as ASCII files are well suited to this. However, there are some features of the CIF syntax that are at variance with the conventional Unix idiom of storing data tags and their associated values within a basic unit of a *line* (*i.e.* a sequence of characters terminated by an end-of-line character code). Although CIFs are built from ASCII-character-populated lines, data values may be placed on a different line from that containing the parent data name (this is almost always the case for looped lists).

5.3.7.1.1. *Description of the ZINC format*

The ZINC format (Stampf, 1994) was developed to transform CIF data into an isomorphous format suitable for manipulation by standard Unix utilities.

The manner in which Unix textual utilities work suggests that an appropriate working format is one in which every individual CIF data value is available within a single-line textual record. The record must also contain information about the context of the value, conveyed through: the name of the data block in which the data value occurs; its associated data name (from which the meaning of the data value is inferred); and for recurrent data (*i.e.* values in a looped list) an indication of the list in which the data value occurs and a counter of its current occurrence in that list. In practice, each record is structured as a data line containing five TAB-separated fields in the order

blockcode *name* *index* *value* *list-id*

where *blockcode* is the name of the CIF data block (the leading `data_` string is omitted); *name* is the data name; *index* is a zero-based index of the number of occurrences of the data name within a loop (with a null value if the data occur outside a loop); *value* is the data value itself; text strings extending over several lines are collapsed into a single line with the replacement of the end-of-line character by the sequence `\n`; and *list-id* is a list identifier, stored as the data name within the list that sorts earliest (because it is a purely syntactic transformation, the utility does not consult a dictionary file for the correct `_list_reference` identifying token).

Comments in the CIF are also stored, to permit regeneration of the original file by an inverse transformation; and because it is often convenient to read such interpolations, especially in interactive activities with CIFs of which the user has no prior knowledge. Comments are stored with a value of '`(`' in the data-name field. They are also numbered (starting from zero) in the index field of the ZINC record.

Most details of the ZINC transformation are illustrated by the simple example in CIF format shown in Fig. 5.3.7.1(*a*). This file transformed to ZINC format would appear on a display terminal as illustrated in Fig. 5.3.7.1(*b*). However, the spacing is deceptive, since typical display terminals convert TAB characters to a variable number of spaces. A more accurate (albeit less legible) representation of the output is given in Fig. 5.3.7.1(*c*).

```
# A simple CIF

data_object

# description of a simple
# polygon
  _name
;
triangle
;
  loop_
      _x _y
      0.0       0.0
      1.0       0.0
      0.0       1.0

      _num_sides 3
```

(*a*)

```
       (        0        # A simple CIF
object (        1        # description of a simple
object (        2        # polygon
object _name             ;\ntriangle\n;
object _x       0        0.0     _x
object _y       0        0.0     _x
object _x       1        1.0     _x
object _y       1        0.0     _x
object _x       2        0.0     _x
object _y       2        1.0     _x
object _num_sides        3
```

(*b*)

```
␣(␣0␣# A simple CIF␣
object␣(␣1␣# description of a simple␣
object␣(␣2␣# polygon␣
object␣␣name␣␣;\ntriangle\n;␣
object␣␣x␣0␣0.0␣␣x
object␣␣y␣0␣0.0␣␣x
object␣␣x␣1␣1.0␣␣x
object␣␣y␣1␣0.0␣␣x
object␣␣x␣2␣0.0␣␣x
object␣␣y␣2␣1.0␣␣x
object␣␣num_sides␣␣3␣
```

(*c*)

Fig. 5.3.7.1. ZINC transformation of CIF. (*a*) is a sample file in CIF format. (*b*) shows the output to a display terminal when this file is transformed by *cifZinc*. (*c*) The same output as (*b*), but with TAB characters represented by special graphical characters.

Note the following points: the data-block name is initially null; each comment line is numbered in sequence from zero; the index field is null for data names that are not within looped lists.

5.3.7.1.2. *ZINC-based utilities*

The purpose of ZINC is to form an intermediate stage in pipeline processes involving Unix tools, and therefore CIF data in ZINC format have only a transitory existence. The ZINC distribution package provides the complementary tools *cifZinc* and *zincCif* required to interconvert formats; and in addition a few sample applications are provided as examples for Unix programmers.

5.3.7.1.2.1. *cifZinc*

cifZinc takes a CIF name as a command-line argument or the CIF itself from standard input and produces a ZINC-format file on standard output. It has one option, '*-c*', which removes comments (which arguably have no place in a CIF).

5.3.7.1.2.2. *zincCif*

zincCif is a Perl script that takes a ZINC-format file (again from standard input or as a name on the command line) and pretty prints the corresponding CIF to standard output. Often, the pipeline `cifZinc a.cif | zincCif > b.cif` produces a more attractive CIF than the original.

5.3.7.1.2.3. *zincGrep*

The shell script *zincGrep* is the utility most requested by those seeing a CIF for the first time. It allows a regular-expression search of a ZINC-format file (or a CIF specified on the command line, which is converted to a ZINC-format file first) and reports the block name, data name, index and value. For example, if the file describing a triangle in the last section were called simple.cif, the command `zincGrep _name simple.cif` would produce

```
object _name            ;\ntriangle\n;
```

5.3.7.1.2.4. *cifdiff*

cifdiff is a C-shell script that takes two CIFs and lists the differences between them. Unlike the standard Unix utility *diff*, which compares files line-by-line, *cifdiff* can determine differences that are independent of reordering and white-space padding.

This script takes each CIF, converts it to a ZINC-format file, then sorts it, first based on the data-block name, then (keeping the loops together) on the data name. It then removes the last field (which is not part of the CIF) and stores the remainder in temporary files. It then runs the standard *diff* program against these reordered temporary files. This is remarkably effective both in finding any differences and in providing the context (it names the block and data name as well as the value) needed to understand the differences.

Fig. 5.3.7.2 illustrates two CIFs that are very different in the presentation of their contents, but have only a small difference of substance in their content. Fig. 5.3.7.3 indicates the output from *cifdiff* that identifies the changed data.

5.3.7.1.2.5. *zb*

zb is a small (less than 200 lines) Tcl/Tk program (Ousterhout, 1994) that provides a simple graphical front end to a ZINC-format file or CIF allowing the user to browse through the contents. Multiple files can be viewed simultaneously, as can multiple data blocks, on any X terminal. *zb* recognizes command-line argument file names in the form *.cif as being in CIF format and converts them to ZINC format automatically.

5.3.7.1.2.6. *zincNl*

zincNl is a Perl script that takes a ZINC file and creates a Fortran-compatible namelist file allowing for easy access to any CIF by Fortran programs without the need for extensive I/O libraries or reprogramming. As with *zb* above, it will automatically convert a CIF to a ZINC-format file if it needs to.

For a more substantial tool providing CIF input functions in Fortran and C, see the discussion below of *CifSieve* (Section 5.3.7.2).

5.3.7.1.2.7. *zincSubset*

zincSubset is another C-shell script which is very short but very useful. It allows a user to generate a custom subset of any ZINC-format file (or CIF) simply by listing the desired data blocks and data names. The script has two file arguments, the first of which specifies a file with regular expressions that specify what is to be included in the subset, and the second of which is the ZINC-format file itself (or standard input). It allows two options: '*-c*' to remove comments and '*-v*' to invert the sense of the search.

It converts the CIF into a ZINC-format file, uses the Unix *grep* program to search through the ZINC-format file for patterns that appear in the regular-expression file and pretty prints the result. For example, the command

```
zincSubset defs cif_core.dic > cifdic.defs
```

will produce a subset of the core CIF dictionary that contains only the names and definitions when the file named 'defs' contains two lines with the TAB-surrounded word '`_name`' and the TAB-surrounded word '`_definition`'.

All the tools listed in this section operate by design in concert with each other, providing the opportunity for generating increasingly complex tools. For example, to generate the Fortran namelist input file with only certain data items, a pipeline of *zincSubset* and *zincNl* will suffice. As more tools are developed, the range of applications will increase many-fold.

```
data_sample
     _title  'scatter graph'
     _description
; A collection of x, y coordinates of points
  drawn in specified colours
;
     loop_
         _x  _y  _colour
          0   0   red
          1   1   red
          2   4   red
          3   9   orange
          4   16  orange
          5   25  orange
      _status complete
```

(*a*)

```
data_sample
_status complete
loop_    _y  _x  _colour
          0   0   red
          1   1   red
          2   2   red
          9   3   orange
          16  4   orange
          25  5   orange
_title  'scatter graph' _description
; A collection of x, y coordinates of points
  drawn in specified colours
;
```

(*b*)

Fig. 5.3.7.2. Two example files in CIF format differing greatly in layout but little in content: (*a*) sample1.cif, (*b*) sample2.cif.

```
% cifdiff sample1.cif sample2.cif
18c18
< sample         _y      2       4
---
> sample         _y      2       2
```

Fig. 5.3.7.3. Output from *cifdiff* comparing files sample1.cif and sample2.cif of Fig. 5.3.7.2. The entries in each line comprise the data-block name, the variable name, the zero-based index of the occurrence of the value in a looped list and the value. Hence it is the *third* value of **_y** in the loop that has changed.

5.3.7.2. *CifSieve*: automatic construction of CIF input functions

Among the utilities described in the ZINC package above was a tool to generate Fortran namelist files. It is a common requirement of applications developers that they should be able swiftly to convert existing programs to read CIF data. While libraries such as *CIFtbx* (Chapter 5.4) and *CIFLIB* (Westbrook *et al.*, 1997) offer very powerful functions for building CIF applications, it can be time-consuming to integrate them with existing software. It is a goal of *CifSieve* (Hester & Okamura, 1998) to enable the rapid creation of new CIF-conversant software by using a CIF dictionary as a template for input data structures.

The *CifSieve* program runs on Unix systems with installed versions of the software utilities and programming languages *bison* or *yacc*, *flex*, Perl (Wall *et al.*, 2000) and C.

5.3.7.2.1. *Overview of the process*

The data names in a CIF are defined in a dictionary written in DDL1 or DDL2 formalism. Therefore, information about the data type and array structure of data variables is already to hand for a software author wishing to determine how to read CIF data into a program's data structures. The *CifSieve* process requires that the programmer augment the relevant CIF dictionary by adding to a copy of the definition of desired items a new attribute, named `_variable_name`, that passes to the application program the name of the associated program variable.

A program *BuildSiv* then reads the augmented dictionary and produces a subroutine capable of reading a CIF and transferring the data items tagged in the augmented dictionary to internal variable storage. The associated data structure is presented in an ancillary file which must be linked to the application program.

CifSieve can produce input subroutines and header or include files for C and Fortran language programs. For C applications, the input subroutine is called *cifsiv_* and is invoked with arguments *cifsiv_(CIF, block)* where *CIF* is the name of the input CIF and *block* is the name of the data block from which data should be read. The data structure is declared in a header file cifvars.h which must be included in subroutines that manipulate the data input from the CIF. For Fortran applications, the input subroutine is also called *cifsiv_*, but takes an additional argument, *blockbeg*, which is the address of the common block containing the input variable names, declared in the include file forcif.inc.

5.3.7.2.2. *The augmented DDL dictionary*

Fig. 5.3.7.4 is an example of the annotations necessary to flag the data names that refer to data items desired to be input from a CIF. The current implementation requires that a copy of the DDL dictionary relevant for the CIF be physically edited to include the new `_variable_name` attribute. The inclusion of such a new attribute will not affect the use of the CIF dictionary for other purposes and by other software.

The definition blocks of data items that are not to be read by the application should be left unchanged.

The value assigned to the `_variable_name` attribute is the name of the variable declared in the application program for storing the input data item. If the items to be input are part of an array (*i.e.* they exist in the CIF as a looped list), the variable name should be supplied as a dimensioned array variable, *e.g.* `atsiteu[1000]` in the example of Fig. 5.3.7.4.

The same attribute (`_variable_name`) may be inserted in DDL1 or DDL2 dictionaries. Separate parsers are supplied for use with either format. When *BuildSiv* is invoked, the parser reads the augmented dictionary and identifies the data items required by the target input subroutine by the presence of a `_variable_name` attribute in the definition block. The definition is read and the relevant values of the type (DDL attribute `_type`), item name (`_name`) and variable name are output in a simple tag–value format and in a standard order. For DDL2 dictionaries, values of `_item_aliases.alias_name` and `_item_linked.parent_name`, if present, are also output. The DDL parser thus transforms and simplifies the dictionary contents.

Where the item-name attribute occurs inside a loop (*i.e.* several data names occur in a single definition block in the dictionary), the variable name for that particular definition block will be given an extra array dimension by *CifSieve*, equal to the number of names in the loop. When a name from this loop is found in a CIF, the value will be read into the respective array location. If an `_item_aliases.alias_name` attribute is present (DDL2), the alias will also be recognized in CIF input files. If this attribute occurs together with looped item names in the domain dictionary, an attempt is made to determine the parent `_item.name` in the loop to which this `_item_aliases.alias_name` refers. This is done within the *BuildSiv* program by examining `_item_linked.parent_name` entries within the same definition block.

Data typing is simplified; the `_item_type.code` values of DDL2 dictionaries are collapsed onto primitive 'numb' or 'char' types. Values of type numb are declared and stored as type double (C) or REAL*8 (Fortran), while values of type char are stored as character arrays char[84] (C) or CHARACTER*84 (Fortran). In consequence, multiple lines of text *cannot* be retrieved with this version of *CifSieve*. Note in particular that values declared as of type 'int' in DDL2 dictionaries will be stored as double-precision real.

```
data_atom_site_aniso_label
    _name                 '_atom_site_aniso_label'
    _category               atom_site
    _type                   char
    _variable_name          mylabel[50]
    _list                   yes

data_atom_site_aniso_U_
    loop_ _name           '_atom_site_aniso_U_11'
                          '_atom_site_aniso_U_12'
                          '_atom_site_aniso_U_13'
                          '_atom_site_aniso_U_22'
                          '_atom_site_aniso_U_23'
                          '_atom_site_aniso_U_33'
    _category               atom_site
    _variable_name          atsiteu[1000]
    _type                   numb

data_reflns_number_
    loop_ _name           '_reflns_number_total'
                          '_reflns_number_observed'
    _category               reflns
    _type                   numb
    _enumeration_range      0:
    _variable_name          reftot

data_refine_ls_extinction_method
    _variable_name          extmet
    _name                 '_refine_ls_extinction_method'
    _category               refine
    _type                   char
    _enumeration_default  'Zachariasen'
```

Fig. 5.3.7.4. Extracts from an augmented DDL1 dictionary (version 1.0 of the core CIF dictionary). The additional `_variable_name` entry is shown in italics.

```
/* These declarations have been automatically
   generated by the cif file input/output function
   generator.  This file should be included in any
   routines that call these functions */
typedef char cifstring[84];
           /* to avoid array complications later */
#define MYLABELMAX 50
#define ATSITEUMAX  1000
typedef double atsiteutype [6];
#ifdef CIFVARDEC
  cifstring errormes;         /* an error message */
  int       errornum;         /* an error number  */
  cifstring mylabel[50];
                     /*data_atom_site_aniso_label*/
  atsiteutype atsiteu[1000];
                        /*data_atom_site_aniso_U_*/
  atsiteutype atsiteuesd[1000];
  cifstring extmet;
               /*data_refine_ls_extinction_method*/
  double reftot [2];         /*data_reflns_number_*/
  double reftotesd [2];
#else
  extern cifstring errormes;  /* an error message */
  extern int       errornum;  /* an error number  */
  extern cifstring mylabel[50];
                     /*data_atom_site_aniso_label*/
  extern atsiteutype atsiteu[1000];
                        /*data_atom_site_aniso_U_*/
  extern atsiteutype atsiteuesd[1000];
  extern cifstring extmet;
               /*data_refine_ls_extinction_method*/
  extern double reftot [2];  /*data_reflns_number_*/
  extern double reftotesd [2];
#endif
```

Fig. 5.3.7.5. Header file cifvars.h for a C application built by *BuildSiv* from the augmented DDL dictionary of Fig. 5.3.7.4.

```
/* A simple example application of the automatically
   generated cifsiv_ function */

#include <stdio.h>
#include "cifvars.h"

main(int argc, char *argv[])
{
  int i;
  char filename[80];
  char block[80];
  printf("Please enter CIF file name: ");
  scanf("%s", filename);
  printf("Please enter data block name ");
  printf("(without data_ prepended): ");
  scanf("%s", block);
  errornum = 0;
  cifsiv_(filename,block);
  if(errornum != 0)  /* an error, we have problems */
    {
    printf("An error occurred in reading the
CIF:\n");
    printf("%s",errormes);
    }
  for(i=0;i<5;i++)
  {
    printf("Atom %d: %s %f %f\n", i, mylabel[i],
            atsiteu[i][0], atsiteu[i][1]);
  }
  printf("Total reflections: %.f\n", reftot[0]);
  printf("Extinction method: %s\n", extmet);
}
```

Fig. 5.3.7.6. An example C program designed to read CIF data as tagged in the augmented DDL dictionary of Fig. 5.3.7.4.

5.3.7.2.3. *Input to a C application program*

When a DDL dictionary *dictfile* has been edited in accordance with the description above, the program *BuildSiv* may be run under a Unix-like operating system with a command of the form

`BuildSiv` *`dictfile ddlversion`*

where *ddlversion* takes the values '1' or '2' to indicate that a DDL1 or DDL2 parser is appropriate. If the option '*-e*' is given before *dictfile*, variable definitions and read capability for standard uncertainty values will be included as well. The name of the variable that will hold the standard uncertainty is the name given by the programmer with the string `esd` appended.

An object file cifsiv.o is produced together with a header file cifvars.h. Some source-code files are also produced as intermediate files in the lexical analysis and parse phases of the software build; these may be deleted. The object file must be linked against the other object files when the application program is compiled and references to the header files must be introduced (generally through C preprocessor `#include` directives) within the application code where access to the imported data structures is required.

Fig. 5.3.7.5 is an example of the header file cifvars.h built when *BuildSiv* reads the augmented dictionary of Fig. 5.3.7.4 with the '*-e*' option to interpret and store standard uncertainties.

The integer variable *errornum* stores a nonzero value if an error occurs in attempting to read a CIF, and an error message is stored in the character array *errormes*, indicating the nature of the problem. Errors generated by the input subroutine *cifsiv_* are not fatal to the parent application program, and will at worst discard the particular loop block or data item affected. The parser operates by discarding CIF data upon encountering an error until it reaches an understandable set of input values. So, for example, if three numbers appear after an item name instead of one, the second two will be ignored after the error variables have been set, and parsing will continue. Similarly, if a serious error occurs within a loop, such as the appearance of an item name not matching an array variable, the entire loop is normally ignored. If a new packet of looped data exceeds the specified array limits, all further data in that loop are ignored.

The *cifsiv_* function has prototype

`void` *`cifsiv_`*`(char*` *`filename`*`, char*` *`blockname`*`)`

and requires pointers to character strings containing the name of the input file and the data-block code from which input is required.

A simple example C application illustrating the use of the *cifsiv_* subroutine is given in Fig. 5.3.7.6.

5.3.7.2.4. *Input to a Fortran application program*

A Fortran program can make use of the C input function generated by *BuildSiv* as long as the compiler used is capable of linking C and Fortran modules. For Fortran applications, the '*-f*' command-line option is used:

`BuildSiv -f` *`dictfile ddlversion`*

A C structure is defined for use within the *cifsiv_* subroutine and an identically constructed Fortran common block is built for use within Fortran routines. The first variable within the common block *must* be passed as an additional argument when the *cifsiv_* function is called. In the current implementation, that variable is

```
C The following common block corresponds to a
C structure defined in the C header, which is written
C to by routine 'cifsiv'. In order to correctly write
C to this common block, 'cifsiv' should be called
C with a *third* argument which will always be
C 'blockbeg'.
     REAL BLOCKBEG
     CHARACTER*84  ERRORMES
     INTEGER ERRORNUM
     CHARACTER*84 mylabel(50)
     REAL*8 atrat(50)
     REAL*8 atratesd(50)
     REAL*8 atsiteu(6,500)
     REAL*8 atsiteuesd(6,500)
     CHARACTER*84 extmet
     REAL*8 reftot(2)
     REAL*8 reftotesd(2)
     COMMON/CIFCMN/BLOCKBEG,ERRORMES,ERRORNUM,mylabel,
    *atrat,atratesd,atsiteu,atsiteuesd,extmet,reftot,
    *reftotesd
```

Fig. 5.3.7.7. Fortran include file forcif.inc for an application built by *BuildSiv* from the augmented DDL dictionary of Fig. 5.3.7.4.

```
      PROGRAM FORGET
      include 'forcif.inc'
      call cifsiv("tbshort.cif","tbal03",blockbeg)
      do i = 1,4
        write(*,*) mylabel(i), atsiteu(1,i),
     *             atsiteu(2,i)
      enddo
      write(*,*) reftot(1)
      write(*,*) extmet
      end
```

Fig. 5.3.7.8. An example Fortran program designed to read CIF data as tagged in the augmented DDL dictionary of Fig. 5.3.7.4.

always called 'BLOCKBEG'. The input subroutine is thus called from within a Fortran program by a line of the type

`CALL CIFSIV(`*FILE*`,` *BLOCK*`,` `BLOCKBEG)`

where *FILE* and *BLOCK* are, respectively, the name of the input file and data block.

Fig. 5.3.7.7 is an example Fortran include file generated by *BuildSiv* and Fig. 5.3.7.8 is an example application incorporating this file. As with the C examples, the CIF data to be read are those specified in the dictionary augmented according to Fig. 5.3.7.4.

It may be noted that the C header file generated by the Fortran implementation of *BuildSiv* (and which is used directly by the C object file produced) is callable by any other C program or subroutine. The Fortran common block is represented by a C structure named *cifcmnptr*, so that the variable names are stored within that structure and must be addressed through the C → operator. That is, an additional C routine compiled in with the Fortran example program of Fig. 5.3.7.7 would refer to the variable holding the value of the input `_refine_ls_extinction_method` as `(char *)cifcmnptr->extmet`.

5.3.8. Tools for mmCIF

The complex relationships between the components of a macromolecular structure at various levels of detail are richly described by the data names in the mmCIF dictionary, but their number and complexity demand more heavyweight tools for proper handling. Input/output for small-molecule or inorganic structures can often be handled by a simple CIF parse and identification of the desired components of one or a few looped data structures. For macromolecules, multiple categories must be loaded simultaneously, and the integrity of relationships between items in the different categories must be properly maintained. For this reason, the most effective tools for mmCIF-based applications have high-level interactions with the mmCIF or related dictionaries, and necessarily involve more complex data manipulations.

In this section are discussed three software systems that are available for work with macromolecular structures: *CIFOBJ* and related libraries, which provide a long-established and complete application program interface (API) to dictionaries and data files; *OpenMMS*, an exciting development allowing abstract data representations (based on the mmCIF dictionary definitions) to be exchanged between applications using an intermediate middleware layer; and *mmLib*, which is a Python toolkit for biomolecular structure applications. These latter two may come closer to the area of domain-specific applications than most of the generic tools we have discussed in this chapter. However, they demonstrate how the abstract data model represented by the mmCIF dictionaries can effectively be imported into a diverse range of programming environments.

5.3.8.1. *CIFOBJ* and related libraries

Early in the development of the mmCIF dictionary, the Nucleic Acid Database at Rutgers University (Berman *et al.*, 1992) created a number of CIF libraries and utilities to underpin data-processing activities. Much of this development work was carried across when the curatorship of the Protein Data Bank was transferred to the Research Collaboratory for Structural Bioinformatics (RCSB; Berman *et al.*, 2002), and the software provides the engine for many of the robust and industrial-strength database operations of these organizations.

CIFLIB (Westbrook *et al.*, 1997) was an early class library, no longer supported, that was developed to provide an API to macromolecular CIF data files and to the associated dictionaries (Chapters 3.6 and 4.5) and underlying dictionary definition language (DDL2) files (Chapter 2.6).

The RCSB Protein Data Bank now distributes object-oriented parsing tools (*CIFPARSE_OBJ*; Tosic & Westbrook, 2000) which fully support CIF data files and their underlying metadata descriptions in dictionaries and DDL2 attribute sets, and a comprehensive library of access methods for data and dictionary objects at category and item level.

The information infrastructure of the Protein Data Bank, built upon these tools, is discussed in Chapter 5.5. All the software produced for this purpose is distributed with full source under an open-source licence, to promote the development of mmCIF tools and to encourage interoperability with other software environments.

5.3.8.2. *OpenMMS*

Object classes represent the first stage in abstracting related data components. By building structured software modules that can manage the small-scale interactions between data components, the programmer can write more succinct code to handle the interactions between much higher-level data constructs. An API then permits third parties to handle the larger-scale objects without any need to know the internal workings of the class library. The next logical step is to present a standard set of 'objects' representing complete logical entities to any programmer for 'plug-and-play' incorporation into new applications.

The Life Sciences Research domain task force of the Object Management Group (OMG, 2001) is concerned with the development of standards for data exchange in biomolecular sciences, and in 2002 approved a macromolecular structure Corba specification. Corba (the common object request broker architecture) is a middleware architecture intended to serve just this purpose of providing access to standard objects representing discrete logical entities suitable for programmatic manipulation. Corba promotes interoperability across networked applications by separating entirely the API from the implementation of the underlying data objects. For applications such as the macromolecular structures database hosted by the Protein Data Bank, the attraction of networked interoperability is that information can be accessed through distributed and federated databases, and can be delivered on demand to any compatible software.

A Corba application comprises an interface definition language (IDL) and an API that together define access to a data structure that encapsulates the abstract representation of the objects and relationships relevant to a particular area of knowledge. In general terms, this data structure may be described as an 'ontology' (Westbrook & Bourne, 2000). The ontology adopted for macromolecular structure (MMS) data was based on the mmCIF dictionary following a submission by the Research Collaboratory for Structural Bioinformatics to a Request for Proposal (Greer, 2000).

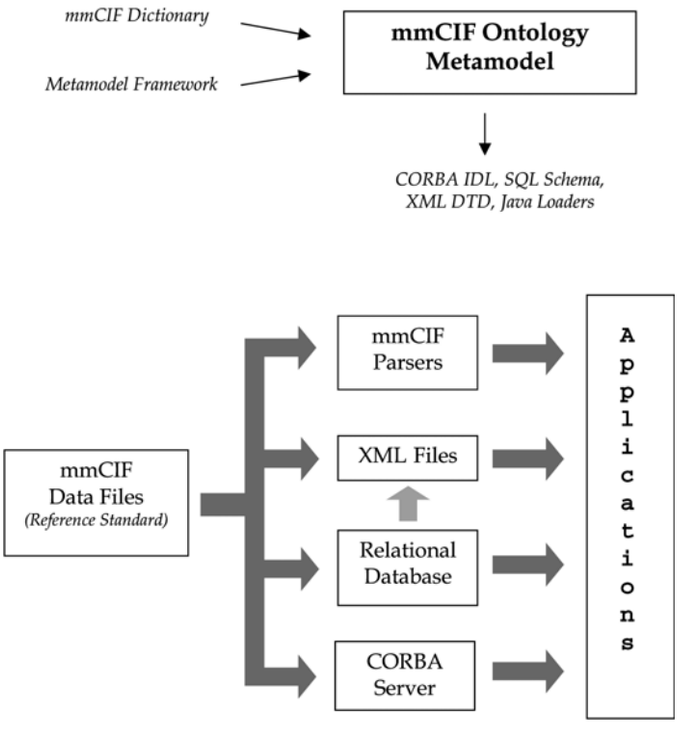

Fig. 5.3.8.1. The *OpenMMS* metamodel and data flow.

5.3.8.2.1. *The OpenMMS toolkit*

In practice, the ontology was developed in a 'metamodel' that combined the definitions and relationships between data items specified in the mmCIF dictionary with a generic metamodel framework. The metamodel extracts the information in the mmCIF dictionary but maintains it in a representation that is independent of the mmCIF STAR or any other file format. The standard building block of the metamodel is an *Entry* object, modelling a single macromolecular structure.

From a suitable metamodel, it then becomes relatively straightforward to generate alternative expressions of the information to suit different access requirements. The *OpenMMS* toolkit (Greer *et al.*, 2002) was built using Java source code to generate a Corba interface, an SQL schema for relational database loading and an XML representation of macromolecular data sets (Fig. 5.3.8.1).

The toolkit contains an mmCIF parsing module capable of direct access to the underlying data archive of mmCIF data files. This is important, because the data files represent a common reference for all the derived representations. Any errors or discrepancies between the expressed forms of the Corba, XML or SQL representations are resolved against the standard mmCIF reference form.

The relational database supporting an SQL-92 compatible interface provides an appropriate API for many applications, particularly ones that require extensive string searches. The close relationship between the mmCIF data model and relational database models has already been described earlier in this volume (Chapter 2.6).

Advantages of the SQL interface are that it provides rapid access direct to the binary data storage representation and that individual components of a data set may be efficiently retrieved without the need to search sequentially through an entire entry.

This efficiency of access and the ability to retrieve individual MMS data elements from a remote server is best realized through the Corba interface, the primary purpose of which is indeed to facilitate such high-performance access.

The bulk exchange of data is addressed through the generation of XML files. XML is a simple, powerful and widely used standard for interchanging data, and its use for transporting macromolecular data obviates the need for target applications to build their own STAR parsers. However, the use of markup tags around every individual data element does make the files much larger than their mmCIF progenitors. This is not an insurmountable problem in large-scale application environments, but it can undermine the effectiveness of XML as a representation mechanism in such applications as web browsers. A possible approach to this could be to define different, less verbose, XML representations and populate these on demand from a database store, either by SQL or XML queries. This is not an approach that the current *OpenMMS* toolkit supports directly.

Fig. 5.3.8.2 is an extract from an XML data file generated from the PDB structure 1xy2. The XML uses a reserved name space *PDBx* conforming to the schema http://deposit.pdb.org/pdbML/pdbx-v0.905.xsd. Data tags map cleanly to the corresponding data names in the mmCIF dictionary formed by concatenating the XML element name with its parent category name. For example, the entry `<PDBx:length_a>27.080</PDBx:length_a>` included in the `<PDBx:cellCategory>` container tag can be directly translated to the corresponding mmCIF data item **`_cell.length_a 27.080`**. CIF data loops are represented by repeated instances of the XML tag representing the corresponding CIF data name (for example, the multiple `<PDBx:audit_author name>` tags are equivalent to a CIF **`loop_ _audit_author.name`** construct). Nonstandard items with a *pdbx_* prefix (*e.g.* `<PDBx:pdbx_description>` in the `<PDBx:entityCategory>` group) refer to private data names in the PDB extension dictionary (Appendix 3.6.2).

5.3.8.3. *mmLib*: a Python toolkit for bioinformatics applications

While the libraries developed for use within the Protein Data Bank provide powerful functionality, their very size and complexity make them inappropriate for some applications. Indeed, considerable effort may be needed to compile the C++ code on non-standard platforms. The *mmLib* toolkit (Painter & Merritt, 2004)

```
<?xml version="1.0" encoding="UTF-8" ?>
<PDBx:datablock  datablockName="1XY2"
 xmlns:PDBx="http://deposit.pdb.org/pdbML/pdbx-v0.905.xsd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation=
    "http://deposit.pdb.org/pdbML/pdbx-v0.905.xsd
        pdbx-v0.905.xsd">
 <PDBx:audit_authorCategory>
  <PDBx:audit_author name="Cooper, S."></PDBx:audit_author>
  <PDBx:audit_author name="Blundell, T.L.">
                                      </PDBx:audit_author>
  <PDBx:audit_author name="Pitts, J.E."></PDBx:audit_author>
  <PDBx:audit_author name="Wood, S.P."></PDBx:audit_author>
  <PDBx:audit_author name="Tickle, I.J."></PDBx:audit_author>
 </PDBx:audit_authorCategory>
 <PDBx:cellCategory>
  <PDBx:cell entry_id="1XY2">
   <PDBx:length_a>27.080</PDBx:length_a>
   <PDBx:length_b>9.060</PDBx:length_b>
   <PDBx:length_c>22.980</PDBx:length_c>
   <PDBx:angle_alpha>90.00</PDBx:angle_alpha>
   <PDBx:angle_beta>102.06</PDBx:angle_beta>
   <PDBx:angle_gamma>90.00</PDBx:angle_gamma>
   <PDBx:Z_PDB>4</PDBx:Z_PDB>
  </PDBx:cell>
 </PDBx:cellCategory>
 <PDBx:citationCategory>
  <PDBx:citation id="primary">
   <PDBx:title>Crystal structure analysis of
               deamino-oxytocin: conformational flexibility
               and receptor binding.</PDBx:title>
   <PDBx:journal_abbrev>Science</PDBx:journal_abbrev>
   <PDBx:journal_volume>232</PDBx:journal_volume>
   <PDBx:page_first>633</PDBx:page_first>
   <PDBx:page_last>636</PDBx:page_last>
   <PDBx:year>1986</PDBx:year>
   <PDBx:journal_id_ASTM>SCIEAS</PDBx:journal_id_ASTM>
   <PDBx:country>US</PDBx:country>
   <PDBx:journal_id_ISSN>0036-8075</PDBx:journal_id_ISSN>
   <PDBx:journal_id_CSD>0038</PDBx:journal_id_CSD>
  </PDBx:citation>
 </PDBx:citationCategory>
 <PDBx:computingCategory>
  <PDBx:computing entry_id="1XY2">
   <PDBx:structure_solution>SHELX</PDBx:structure_solution>
   <PDBx:structure_refinement>SHELX-76
                              </PDBx:structure_refinement>
  </PDBx:computing>
 </PDBx:computingCategory>
 <PDBx:database_2Category>
  <PDBx:database_2 database_id="PDB" database_code="1XY2">
                                        </PDBx:database_2>
 </PDBx:database_2Category>
 <PDBx:entityCategory>
  <PDBx:entity id="1">
   <PDBx:type>polymer</PDBx:type>
   <PDBx:src_method>man</PDBx:src_method>
   <PDBx:pdbx_description>OXYTOCIN</PDBx:pdbx_description>
   <PDBx:formula_weight>978.189</PDBx:formula_weight>
   <PDBx:pdbx_number_of_molecules>1
                           </PDBx:pdbx_number_of_molecules>
  </PDBx:entity>
  <PDBx:entity id="2">
   <PDBx:type>water</PDBx:type>
   <PDBx:src_method>nat</PDBx:src_method>
   <PDBx:pdbx_description>water</PDBx:pdbx_description>
   <PDBx:formula_weight>18.015</PDBx:formula_weight>
   <PDBx:pdbx_number_of_molecules>7
                           </PDBx:pdbx_number_of_molecules>
  </PDBx:entity>
 </PDBx:entityCategory>
```

Fig. 5.3.8.2. Sample XML output from the *OpenMMS* XML generator. Lines have been omitted or wrapped to fit the present column width.

addresses this by supplying a library of object-oriented routines implemented in Python (van Rossum, 1991) that are designed to integrate with existing or new applications in an easy way.

The objective of *mmLib* is to build a support platform to handle the increasingly rich data about macromolecular structure

Table 5.3.8.1. *The modules provided by the mmLib toolkit*

mmLib.mmCIF	mmCIF parser
mmLib.PDB	PDB format parser
mmLib.Library	Base chemical library
mmLib.Extensions.CCP4Library	Data retrieval from CCP4 monomer library
mmLib.Elements	Chemical data for elements
mmLib.AminoAcids	Chemical data for amino acids
mmLib.NucleicAcids	Chemical data for nucleic acids
mmLib.Structure	Macromolecular structure model
mmLib.GLViewer	OpenGL visualizer

```
import mmLib
from mmLib.FileLoader import LoadStructure, SaveStructure
    struct = LoadStructure(
                fil = cif,
                format = "PDB",
                build_properties = ("no_bonds",) )
    SaveStructure(
        fil = pdb,
        structure = struct,
        format = "CIF")
```

Fig. 5.3.8.3. A snippet of code illustrating mmCIF/PDB file format conversion with the *mmLib* toolkit.

available to structural biologists. Not only do applications need to be able to handle atomic positions and build appropriate three-dimensional structure representations; but links to and integration with information on sequence, homologous structures, and biochemical, genetic and medical form and function are also demanded from individual program systems. Since much of these data are available from external databases in a variety of formats, *mmLib* will not be restricted to the handling of files in a single format. Its initial release provides support for mmCIF, for the PDB format files that historically have been used for representation of macromolecular structures (Westbrook & Fitzgerald, 2003) and for the MTZ format used by the *CCP4* program suite (Collaborative Computational Project, Number 4, 1994).

Table 5.3.8.1 lists the main modules in the current release. *mmLib.mmCIF* and *mmLib.PDB* are read/write parsers for mmCIF and PDB format files, respectively, which handle file input and output in these formats, and provide support for inspection or modification of such file formats. They are typically used in conjunction with the *mmLib.FileLoader* component to populate the *mmLib.Structure* internal representation of the macromolecular structure. The high-level abstraction of such functionality allows for very succinct programmatic constructs. Fig. 5.3.8.3 illustrates this with a program snippet that (apart from the necessary system calls for file management) achieves the conversion of an mmCIF input file to a PDB format representation. This is sufficiently robust and lightweight to act as an input filter to software already designed for handling PDB format files.

mmLib.Structure represents the internal representation of a molecular structure and is implemented as an object hierarchy with four basic object classes: *Structure*, *Chain*, *Fragment* and *Atom*. The *Fragment* class has subclasses *AminoAcidResidue* and *NucleicAcidResidue*. In order to build a complete representation of a structure, the toolkit may need to load data from an input mmCIF or PDB format file, and also from standard data sets of properties of individual monomers and chemical elements; these standard libraries of chemical properties are provided by the *mmLib.Library* module. The core *mmLib* source includes a limited library of such chemical properties (accessible through the subclasses *mmLib.Elements*, *mmLib.AminoAcids* and *mmLib.NucleicAcids*)

and also provides support for the extensive CCP4 monomer library through the *mmLib.Extensions.CCP4Library*. The naming of this class expresses the intention that other standard data sources should be made accessible in the same way.

The CCP4 monomer library is in fact included with the software as a directory tree of small files in mmCIF format, which are loaded into the *Structure* object through the normal use of the toolkit's mmCIF parser.

mmLib.GLViewer is a module provided to support visualization programs using the OpenGL graphics environment. Although it does not by itself provide a stand-alone viewer, it can be incorporated into many common graphics application building environments. An example molecular viewer, *mmView*, is provided with the distribution as an example of an application using the GTK graphical user interface, a popular toolkit in Linux.

5.3.9. Concluding remarks

CIF is a domain-specific format that cannot attract the number of programmers that generic formats such as XML do. In spite of this, there is an impressive collection of programs available to support activities at many levels, from the single-line shell script needed to search for some desired content in a collection of CIFs, to the industrial-scale activities of major databases and publishing houses. As many examples as possible of the programs discussed in this chapter have been collected on the IUCr web site (http://www.iucr.org/iucr-top/cif/software). It is hoped that the contributions described here will inspire future generations of programmers to contribute to a growing and increasingly robust software collection to make the use of CIFs ever easier and more fruitful.

I am immensely grateful for the assistance, cooperation and involvement of the community of software authors who have contributed to this chapter in one way or another, and to all the programmers and developers who have been active through the cif-developers discussion list of the IUCr (http://www.iucr.org/iucr-top/lists/cif-developers) and in private discussions.

References

Allen, F. H. (2002). *The Cambridge Structural Database: a quarter of a million crystal structures and rising*. *Acta Cryst.* B**58**, 380–388.

Allen, F. H., Johnson, O., Shields, G. P., Smith, B. R. & Towler, M. (2004). *CIF applications. XV. enCIFer: a program for viewing, editing and visualizing CIFs. J. Appl. Cryst.* **37**, 335–338.

Berman, H. M., Battistuz, T., Bhat, T. N., Bluhm, W. F., Bourne, P. E., Burkhardt, K., Feng, Z., Gilliland, G. L., Iype, L., Jain, S., Fagan, P., Marvin, J., Padilla, D., Ravichandran, V., Schneider, B., Thanki, N., Weissig, H., Westbrook, J. D. & Zardecki, C. (2002). *The Protein Data Bank. Acta Cryst.* D**58**, 899–907.

Berman, H. M., Olson, W. K., Beveridge, D. L., Westbrook, J., Gelbin, A., Demeny, T., Hsieh, S.-H., Srinivasan, A. R. & Schneider, B. (1992). *The Nucleic Acid Database: a comprehensive relational database of three-dimensional structures of nucleic acids. Biophys. J.* **63**, 751–759.

Bernstein, H. J. (1998). *cif2cif. CIF copy program.* http://www.iucr.org/iucr-top/cif/software/ciftbx/cif2cif.src/.

Bernstein, H. J. & Hall, S. R. (1998). *CIF applications. VII. CYCLOPS2: extending the validation of CIF data names*. *J. Appl. Cryst.* **31**, 278–281.

Bluhm, W. (2000). *STAR (CIF) parser*. http://pdb.sdsc.edu/STAR/index.html.

Brown, I. D., Zabobonin, A. & Holt, B. (2004). *beCIF. Browser and editor for CIF.* Private communication.

Collaborative Computational Project, Number 4 (1994). *The CCP4 suite: programs for protein crystallography. Acta Cryst.* D**50**, 760–763.

Edgington, P. R. (1997). *HICCuP: High-Integrity CIF Checking using Python*. Cambridge: Cambridge Crystallographic Data Centre.

Greer, D. S. (2000). *Macromolecular structure RFP response*. Revised submission. http://openmms.sdsc.edu/OpenMMS-1.5.1_Std/openmms/docs/specs/lifesci_00-11-01.pdf.

Greer, D. S., Westbrook, J. D. & Bourne, P. E. (2002). *An ontology driven architecture for derived representations of macromolecular structure. Bioinformatics*, **18**, 1280–1281.

Hall, S. R. (1993). *CIF applications. III. CYCLOPS: for validating CIF data names*. *J. Appl. Cryst.* **26**, 480–481.

Hall, S. R., Allen, F. H. & Brown, I. D. (1991). *The Crystallographic Information File (CIF): a new standard archive file for crystallography. Acta Cryst.* A**47**, 655–685.

Hall, S. R. & Bernstein, H. J. (1996). *CIF applications. V. CIFtbx2: extended tool box for manipulating CIFs*. *J. Appl. Cryst.* **29**, 598–603.

Hall, S. R. & Sievers, R. (1993). *CIF applications. I. QUASAR: for extracting data from a CIF*. *J. Appl. Cryst.* **26**, 469–473.

Hester, J. R. (2006). *A validating CIF parser: PyCIFRW. J. Appl. Cryst.* **39**, 621–625.

Hester, J. R. & Okamura, F. P. (1998). *CIF applications. X. Automatic construction of CIF input functions: CifSieve*. *J. Appl. Cryst.* **31**, 965–968.

Knuth, D. E. (1986). *The TEXbook*. *Computers and Typesetting*, Vol. A. Reading, MA: Addison-Wesley.

McMahon, B. (1993). *ciftex: translation utility from CIF to TEX.* ftp://ftp.iucr.org/pub/ciftex.tar.Z.

McMahon, B. (1998). *vcif: a utility to validate the syntax of a Crystallographic Information File.* http://www.iucr.org/iucr-top/cif/software/vcif/index.html.

OMG (2001). *Life Sciences Research Domain Task Force.* http://www.omg.org/lsr/.

Ousterhout, J. K. (1994). *Tcl and the Tk toolkit*. Reading, MA: Addison-Wesley.

Painter, J. & Merritt, E. A. (2004). *mmLib Python toolkit for manipulating annotated structural models of biological macromolecules. J. Appl. Cryst.* **37**, 174–178.

Patel, A. J. (2002). *Yapps: Yet Another Python Parser System.* http://theory.stanford.edu/∼amitp/yapps/.

Rossum, G. van (1991). *Python programming language*. http://www.python.org.

Spadaccini, N. & Hall, S. R. (1994). *Star_Base: accessing STAR File data*. *J. Chem. Inf. Comput. Sci.* **34**, 509–516.

Stampf, D. R. (1994). *ZINC: galvanizing CIF to work with UNIX.* Brookhaven: Protein Data Bank.

Toby, B. H. (2003). *CIF applications. XIII. CIFEDIT, a program for viewing and editing CIFs*. *J. Appl. Cryst.* **36**, 1288–1289.

Tosic, O. & Westbrook, J. D. (2000). *CIFParse. A library of access tools for mmCIF.* Reference guide. http://sw-tools.pdb.org/apps/CIFPARSE-OBJ/cifparse/index.html.

Wall, L., Schwartz, R. L., Christiansen, T. & Orwant, J. (2000). *Programming Perl*, 3rd ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc.

Westbrook, J. D. & Bourne, P. E. (2000). *STAR/mmCIF: an ontology for macromolecular structure*. *Bioinformatics*, **16**, 159–168.

Westbrook, J. & Fitzgerald, P. (2003). *The PDB format, mmCIF formats and other data formats. Structural bioinformatics*, edited by P. E. Bourne & H. Weissig, pp. 161–179. Hoboken, NJ: John Wiley & Sons, Inc.

Westbrook, J. D., Hsieh, S.-H. & Fitzgerald, P. M. D. (1997). *CIF applications. VI. CIFLIB: an application program interface to CIF dictionaries and data files. J. Appl. Cryst.* **30**, 79–83.

Westrip, S. P. (2004). *printCIF for Word*. http://www.iucr.org/iucr-top/cif/software/printCIFforWord/index.html.

Winn, M. (1998). *cif.el: an Emacs mode for CIF*. Daresbury Laboratory, Warrington, England.