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.