CIF APPLICATIONS

Authors of any software that reads, writes or validates CIF data are invited to contribute to this series. Authors should state clearly when submitting a manuscript to a Co-editor that the paper should be included as part of the CIF Applications series. An appropriate series number will be assigned by the Editorial Office.

J. Appl. Cryst. (1998). **31**, 965–968

CIF Applications. X. Automatic Construction of CIF Input Functions: *CifSieve*

J. R. Hester*† and F. P. Okamura *at National Institute for Inorganic Materials, Namiki 1-1, Tsukuba, Ibaraki 305, Japan. E-mail: jrh@nirim.go.jp*

(*Received 10 February 1998; accepted 30 June 1998*)

Abstract

A software package for reading a list of CIF data into user-specified variable names in the DDL domain dictionary is described. The customized function generated by this process provides detailed error reporting and may be called from C or Fortran programs. The package is small, simple to install and fast. It runs on variants of the Unix operating system that have the following utilities available: Bison/Yacc, Flex, Perl and C.

1. Introduction

Since the Crystallographic Information File (CIF) was originally proposed (Hall *et al*., 1991) for exchange and storage of crystallographic data, a range of software tools have been developed for manipulating this format (Hall & Bernstein, 1996; Westbrook *et al*., 1997). These tools provide functions for checking, reading and writing CIFs, and reduce the complexity of handling the flexible CIF and dictionary definition language (DDL) syntax for programmers wanting to create a CIF-literate application.

Even so, when using such tools, a programmer is still required to specify the CIF name and destination data structure for every required item. This operation is both time consuming and prone to transcription errors, particularly when a large amount of data is involved. A simpler access route to CIF data is needed for small in-house applications in order to encourage the wider crystallographic community to use CIF in place of less portable fixed formats.

It is possible to reduce substantially the programming time required to write an input CIF interface by using the DDL domain dictionary as a simple template for the customized software. This leads to the rapid creation of new CIF-conversant software which can be easily added to existing programs.

In the following description, 'domain dictionary' refers to a CIF dictionary in DDL format.

2. Applying *CifSieve*

The program *CifSieve* may be used to generate a customized CIF reader for addition to existing software by the following two basic steps.

(i) Edit an existing domain dictionary file by inserting a new attribute and value `_variable_name` *variable-name* in the definition blocks of those items the application program needs to read from the CIF. See Fig. 1 for an example of how this is done. The definitions of items that do not need to be read should be left unchanged. Note that the addition of `_variable_name` does not alter other uses of this dictionary. *variable-name* is the name of the variable declared in the application program for storing the read data item. If the items to be read are part of an array (*i.e.* they exist in the CIF as a looped list) then *variable-name* should be entered as a dimensional variable, *e.g.* FCAL(2000).

(ii) For C applications the program *BuildSiv* is used to create an object file for the function `cifsiv_` and a separate header file `cifvars.h`. These are invoked in an application program as `cifsiv_`(*CIFfilename*, *blockname*) to read and store items (as tagged in the domain dictionary) from the CIF named *CIFfilename* and the data block named *blockname*. The header file `cifvars.h` is included in subroutines which manipulate the data input from the CIF. Examples of a modified dictionary file are shown in Fig. 1. This caused *BuildSiv* to generate a `cifsiv_` object file which may be

```
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. 1. A section of an edited DDL file.

† Current address: ANBF, KEK-PF, Oho 1-1, Tsukuba, Ibaraki 305, Japan. E-mail: jrh@anbf2.kek.jp.

invoked as shown in Fig. 2, and the header file `cifvars.h` as shown in Fig. 3.

For Fortran applications the program *BuildSiv* is used to create an object file containing the function `cifsiv_` and an include file `forcif.inc` specifying the common block containing the input variable names. `cifsiv_` is applied in a Fortran program as `call cifsiv(`*CIFfilename*, *blockname*, *blockbeg*) with an additional third argument, which is the address of the beginning of this common block (this is described further below).

3. Design

The *CifSieve* package comes in two parts: the DDL parser program for the domain dictionary and the main *BuildSiv* program responsible for constructing the `cifsiv_` function

```
/* 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 (without data_ prepended): ");
  scanf("%s", block);
  errornum = 0;
  cifsiv_(filename,block);
  if(errornum != 0)      /* an error, we have problems */
  {
    printf("An error occured while reading the CIF file:\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. 2. An example C application program.

```
/*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. 3. The generated C variable declarations file `cifvars.h`.

source from the DDL parser output. *CifSieve* relies heavily on freely available software, particularly from the Free Software Foundation (Stallman, 1993).

3.1. *DDL parser*

Separate DDL parsers are provided for the original DDL specification (hereafter DDL1) (Hall & Cook, 1995) and for DDL2 (Westbrook & Hall, 1995). The parsers are automatically constructed from a restricted STAR grammar specification in Bison format (Donnelly & Stallman, 1995), using a Flex-generated lexical analyser (Paxson, 1995). The parser scans the dictionary and if the `_variable_name` attribute occurs within a dictionary definition, a flag is set, and when reading of that definition finishes, variable type (the value of definition attribute `_type`), item name (attribute `_name`) and variable name are output in a simple tag-value format and in a standard order. For DDL2 domain dictionaries, values of `_item_aliases.alias_name` and `_item_linked.parent_name` entries, if present, are also output. The DDL parser thus transforms and simplifies the dictionary contents.

If the `_name`(DDL1)/`_item.name`(DDL2) attribute occurs inside a loop, that is, a number of names appear in one definition block, the variable name for that particular definition block will be given an extra array dimension by *CifSieve*, equal to the number of items in the loop. When a CIF name from this loop is found in a CIF file, 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 that loop to which this `_item_aliases.alias_name` refers. This is done within *BuildSiv* by examining `_item_linked.parent_name` entries within the same definition block.

3.2. *Building* `cifsiv_`*: BuildSiv*

The *BuildSiv* program, which is a shell-like script written in Perl (Wall *et al*., 1996), creates four source files, which together describe the final input function. One file, `cifvars.h`, is a list of declarations for the variables which will contain the CIF data. The second file contains a short C-language wrapper function which calls the parser generated by Bison and Flex. The next two files, `cifsiv.y` and `cifsiv.lex`, are specification files for the parser and its lexical analyser, which are constructed from these files by Bison and Flex, respectively.

BuildSiv first calls the DDL parser described above, which returns a simplified version of the domain dictionary. The contents of this file are read and a Flex specification file constructed for lexical analysis of the CIF file. Variable declarations are output to file `cifvars.h` as each variable name and type is encountered.

At present, the correspondence between a DDL1 domain dictionary `_type` attribute and the compiled function variable type is as follows: `numb` becomes `double` (C) or `REAL*8` (Fortran). `char` becomes `char[84]` (C) or `CHARACTER*84` (Fortran). Multiple lines of text cannot be retrieved using the present version of *CifSieve*.

Dictionaries written using DDL2, such as the mmCIF dictionary, allow internal definition of the meaning of values for `_item.type` tags. These type definitions are not presently parsed by DDL2; instead, the types defined in mmCIF version 1.0.00 are recognized and mapped such that all numbers (`float`, `int`) are treated identically to the DDL1 `numb` type, and all character types become no more than 84 characters long.

The Bison parser specification file, `cifsiv.y`, is then composed. The grammar is specified such that, if a target item name is encountered outside a loop, the item value is explicitly copied to the variable name. Looped syntax is more complex. A loop is divided into a `looptop` and `loopbottom`, where `looptop` is a list of looped item names and `loopbottom` is a list of their values. When one of the target CIF item names is found in the `looptop` section, a pointer is set to the first entry in the user array variable corresponding to that CIF item. Then, when the bottom part of the loop is being input, data values corresponding to that data item are copied into the position specified by this pointer, and the pointer incremented to point to the next location. If the programmer-specified variable name does not contain an array specification, it will not be included in this section of the grammar specification file, and an error message will therefore be generated if it is encountered inside a loop during CIF file input.

BuildSiv then runs Bison and Flex to generate the C source code for the parser. Finally, the C wrapper program is created and compiled together with the parser source code to produce the final object file containing the `cifsiv_` function.

If *BuildSiv* is called with the `-e` option, variable declarations for e.s.d.s are also output and e.s.d.s for requested items are

```
C The following common block corresponds to a structure defined
C in the C header, which is written to by routine 'cifsiv'.
C In order to correctly write to this common block, 'cifsiv'
C should be called with a *third* argument which will always be
C 'blockbeg'.
      REAL BLOCKBEG
      CHARACTER*84  ERRORMES
      INTEGER ERRORNUM
      CHARACTER*84 mylabel(50)
      REAL*8 atsiteu(6,1000)
      REAL*8 atsiteuesd(6,1000)
      CHARACTER*84 extmet
      REAL*8 reftot(2)
      REAL*8 reftotesd(2)
      COMMON/CIFCMN/BLOCKBEG,ERRORMES,ERRORNUM,mylabel,atsiteu,
     *atsiteuesd,extmet,reftot,reftotesd
```

Fig. 4. The example Fortran include file `forcif.inc`.

read in if they appear after numerical values in the CIF file. Variable names for the e.s.d.s are created by appending the letters 'esd' to the user variable name.

The generated function is robust relative to syntax and type errors within the CIF file. If an error occurs, variable `errornum` is set to a nonzero value and an error message is inserted into string `errormes`. These variables are declared together with the user variables. The parser then discards CIF data 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. Similary, 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, further data in that loop are ignored.

3.3. *Fortran interface*

When the `-f` option is given to *BuildSiv*, a Fortran interface is generated. The automatically generated file `forcif.inc` defines a common block containing the user-specified variable names.

The Fortran interface is implemented by defining both a C structure, for use within `cifsiv_`, and an identically constructed Fortran common block, for use by application programs. When `cifsiv_` is called from Fortran with the first element of this common block, `BLOCKBEG`, as a third argument, the `cifsiv_` function receives a pointer to that argument and consequently to the beginning of the Fortran common block. This pointer value is used as the address of the beginning of the C structure and item values are thus read into the proper positions within this structure by `cifsiv_`. After `cifsiv_` returns, Fortran application programs can read variable values from this common block.

```
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. An example Fortran program using `cifsiv`.

4. Example

The following example shows the various stages of processing of the edited dictionary. Fig. 1 shows an edited dictionary with `_variable_name` attributes. Fig. 2 shows a short test program demonstrating the use of the `cifsiv_` function. Fig. 3 shows the C header file generated, `cifvars.h`, which should be included in any routines using the variables.

The Fortran include file, together with a simple program, are given in Figs. 4 and 5. The `cifvars.h` file generated in this case (not shown) is different to that in Fig. 2, as all variables are now defined as members of a structure.

5. Availability

The program, with installation and detailed operating instructions, is freely available by ftp at ftp://anbf2.kek.jp/pub/cif/cifsieve_1.2.tar.gz. GNU software is available from the Free Software Foundation at ftp://prep.ai.mit.edu/pub/gnu.

The authors are grateful for helpful discussions with Syd Hall and thank Timo Vaalsta for conceiving the idea behind the Fortran interface. We owe a debt of gratitude to the programmers who have contributed to the GNU project.

References

Donnelly, C. & Stallman, R. (1995). *The Bison Manual*. Free Software Foundation, 59 Temple Place Suite 330, Boston MA 02111, USA, http://www.fsf.org.

Hall, S. R., Allen, F. H. & Brown, I. D. (1991). *Acta Cryst.* A**47**, 655–685.

Hall, S. R. & Bernstein, H. J. (1996). *J. Appl. Cryst.* **29**, 598–603.

Hall, S. R. & Cook, A. P. F. (1995). *J. Chem. Inform. Comput. Sci.* **35**, 819–825.

Paxson, V. (1995). *The Flex Manual*. Free Software Foundation, 59 Temple Place Suite 330, Boston MA 02111, USA, http://www.fsf.org.

Stallman, R. (1993). *The GNU Manifesto*. Free Software Foundation, 59 Temple Place Suite 330, Boston MA 02111, USA, http://www.fsf.org.

Wall, L., Christiansen, T. & Schwartz, R. L. (1996). *Programming Perl*, 2nd ed. California, USA: O'Reilly.

Westbrook, J. & Hall, S. R. (1995). *A Dictionary Description Language for Macromolecular Structure*, Draft DDL V2.1.0. IUCr-COMCIFS, Chester, England.

Westbrook, J. D., Hsieh, S. & Fitzgerald, P. M. D. (1997). *J. Appl. Cryst.* **30**, 79–83.