

On the Use of Specifications of Binary File Formats for Analysis and Processing of Binary Scientific Data [★]

Alexei Hmelnov^{1,2}[0000–0002–0125–1130] and Tianjiao Li³[0000–0002–1037–2452]

¹ Matrosov Institute for System Dynamics and Control Theory of Siberian Branch of Russian Academy of Sciences, 134 Lermontov st. Irkutsk, Russia

`hmelnov@icc.ru`

² Institute of Mathematics, Economics and Informatics, Irkutsk State University, Gagarin Blvd. 20, Irkutsk, Russia

³ ITMO University, 49 Kronverksky Pr., St. Petersburg, Russia

`litianjiao@mail.ru`

Abstract. The data collected during various kinds of scientific research may be represented both by well known binary file formats and by custom formats specially developed for some unique device. While thorough understanding of the file format may be required for the former case of the well known format, for the latter case of custom formats it is of critical importance. For the custom formats usually only few people know how they are organized, and this expertise can easily be lost.

We have developed the language FlexT (Flexible Types) for specification of binary data formats. The language is declarative and designed to be well understood for human readers. Its main elements are the data type declarations, which look very much like the usual type declarations of the imperative programming languages, but are more flexible. The primary purpose of the language FlexT development is to be able to view the binary data and to check that the data conform to the specification, and that the specification conforms to the data samples available. As a result of the tests we can be sure, that the specification is correct. The FlexT specifications doesn't contain any surplus information besides from that about the file format. They are compact and human readable. We have also developed the algorithm for data reading code generation from the specification.

In the report we'll consider some FlexT language details and the experience of its application to specification of some scientific data formats.

Keywords: Specifications of binary data formats · Declarative language · Scientific data life-cycle.

[★] The work was carried out with financial support of Russian Foundation for Basic Research, grant No. 18-07-00758-a. Some results were obtained using the facilities of the Centre of collective usage "Integrated information network of Irkutsk scientific educational complex"

1 Introduction

In the field of natural sciences the outcomes of some research are usually represented by the articles, which summarize the main results of the analysis of the collected data. But the results of the analysis depend also on the methods chosen, their parameters, and some other subjective factors, so the other researchers are interested in the access to the data (both raw original and processed) to be able to check the results, try their own ways of the data analysis, or compare the data to their own ones. The concept of scientific data life-cycle should meet this demand.

So, it becomes not enough to just obtain the data, process them and write some articles using the results of the processing. It is also required to share the data with other researchers. These researchers may be not only our contemporaries but also our descendants, living in a few decades from now. Our descendants, we hope, will have more advanced devices, computers and software. But they still will not be able to measure the values of physical quantities for previous periods (say, several years ago). So the data, that we have stored for them, may become very valuable. Let us consider the possible ways of the data representation, compare their features, and describe some approaches, that should simplify data sharing.

2 Options for choosing the format of scientific data

When deciding on the data representation it is required to select between several groups of options. We may consider the groups as a coordinates in the space of possible file formats. Let us consider the dimensions and their possible values.

The first classification is by the level of customization:

- well-known file formats with ready to use data processing libraries and software (JPEG,TIFF,DBF,CSV,SHP,...);
- data exchange frameworks with the data format setup/data processing/code generation libraries, software;
- "universal" formats with the data format setup/data processing/code generation libraries, software;
- custom file formats.

So it is required to select between the well-known file formats, which may already have the ready to use libraries for their processing; the data exchange frameworks and "universal" formats, which require some additional information about the particular data structures, but provide a lot of ready to use services for their handling and the custom file formats, which will require to write all the software from scratch. Since the existing file formats may impose some unacceptable limitations on the data structures or have too large memory overhead, the choice here is far from being obvious. The "universal" file formats will be addressed in more detail in the next sections.

The second classification is by the file format usage: there exist internal and exchange file formats. The internal file formats are designed to maximize the efficiency and ease of processing by a specific software. They may contain some auxiliary data structures (e.g. indices) and be very complex. The internal file formats are usually binary, unless the files are very small. The exchange file formats should be rather simple and easily understandable for other programmers, but they may be not very well-suited for everyday work (e.g. data editing).

The next option to consider when selecting the data representation is whether to use a text-based or a binary data representation. The binary data formats are much more space- and time-efficient, than the text-based ones. The main disadvantage of the binary data is that they look opaque to the users and it is hard to control their contents with a "naked eye". That's why programmers nevertheless often prefer to use text formats, and among them the XML-based ones are of great popularity in spite of the fact that it becomes impossible for a human being to comprehend the extremely large text files. The language FlexT can make binary data transparent.

3 The "Universal" formats

The "Universal" formats allow you to store a wide (but still limited) range of information. They are always accompanied by some software libraries and utilities for handling the files of the format.

Some examples:

- XML (eXtensible Markup Language) is a text based format family, they may use XML schemata, various utilities and editors, libraries/code generation software to simplify data processing. Various file format projects claim to be the binary version of the XML format, but no proposal has been accepted as a binary XML standard.
- Hierarchical Data Format (HDF) [1] is a set of file formats (HDF4, HDF5) designed to store and organize large amounts of data. Originally developed at the National Center for Supercomputing Applications, the formats are now supported by the HDF Group. It has libraries for C,C++,Java, MATLAB, Scilab, Octave, Mathematica, IDL, Python, R, Fortran, and Julia. The main goal of its development is to create portable scientific data format. HDF files are self-describing allowing an application to interpret the structure and contents of a file without an external specification. One HDF4 file can hold a mix of related objects which can be accessed as a group or as individual objects. Users can create their own grouping structures called "vgroups". HDF5 simplifies the file structure to include only two major types of object:
 - datasets, which are multidimensional arrays of a homogeneous type;
 - groups, which are container structures that can hold datasets and other groups.
- Protocol Buffers (by Google) [2] is a method of serializing structured data. The method involves an interface description language that describes the structure of some data and a program that generates source code from that

description for writing or reading a stream of bytes that represents the structured data. In contrast to HDF the resulting files are not self-contained and require the structure specification (or, better, the code generated from it) to read the files, corresponding to this specification.

”Universal” data formats are often used to represent scientific data. Not to mention XML, which is used everywhere, HDF files, for example, are used for representation of remote sensing data, while the Protocol Buffers are the main way to represent the weights of deep neural networks. The main reason for this is the availability of tools and libraries that simplify the development and processing of ”Universal” formats at the cost of following their prescripts. Our approach makes it possible to do nearly the same with an almost arbitrary binary file format.

4 Binary file format specifications in FlexT

The main goal of the language FlexT is to provide the instrument, that can help us to explore and understand the contents of the binary files, for which there exists a specification of their format. It can also help to check whether the format specification is correct using the samples of data in this format. FlexT specifications can be used to check, view, process, and document both well-known and custom file formats.

Our experience shows that the vast majority of format specifications written in natural language contain errors and ambiguities, which can be detected and fixed by trying to apply various versions of the specification to the sample data in order to find the correct variant of understanding of the format description.

The information about a file format may also be obtained from the source code of a program that reads or writes the data. But the data access code contains a lot of unessential details about a specific way to read or write data and it is usually intermixed with the other code, that somehow processes the data. So, the resulting FlexT specification, which leaves out all the surplus details, will be much more concise and of much higher quality than the source code.

In contrast to many other projects intended for specification of binary file formats, the FlexT specifications are human-readable, and it is possible to consider them as a concise notation for representation of the information about a binary file format.

We also have a successful experience of reverse engineering of some file formats using just the samples of data without any description or source code.

The format specifications are also required to write a correct program, that should work with the files of the format. Because the FlexT language data types look similar to that of imperative languages, it is possible to immediately use some parts of specification to declare the data types, constants, and so on, which are required to write the data processing code. Anyway the process of writing the code manually is still time-consuming and error-prone and we have implemented the code generator, which can automatically produce the data reading code in imperative languages from the FlexT specifications.

By now we have implemented the code generation for the most widely used FlexT data types, but some complex types like that used in specifications of machine instructions' encoding are not supported yet.

5 Features of the FlexT language

The major part of the information about a file format is represented in the FlexT specification by the data type declarations. In contrast to the data types of imperative programming languages, the FlexT data types can contain data elements, the size of which is determined by the specific data represented in the format. Thus we can say that the types flexibly adjust to the data. It explains the name of the language FlexT (from Flexible Types). After defining the data types, it is required to specify the placement in memory of some data elements which have some of these types. Such top-level data elements by analogy with imperative programming languages we will call *variables* (although they represent immutable data).

The language syntax was chosen to be well-understandable by human reader. When designing some of the languages discussed in the review, such as the EAST language, this task was explicitly formulated in the requirements for development. The language FlexT was discussed in more details in [3]. Let us briefly consider the main features of the language to the extent that will make the article self-contained.

5.1 Dynamic and static data types

By *static* data types we will understand the data types analogous to the traditional types of procedural programming languages. Their distinctive feature is that the size of a data element of this type and the internal placement of its constituent parts are determined at the time of compilation and do not depend on specific data. In procedural programming languages that are actually used at present, the composite data types can contain only the static elements.

The size of a data element of *dynamic* type and the internal placement of its sub-elements may depend on specific data, which it represents. Hereinafter the word "dynamic" will be used in this sense. String constants are an example of dynamic types in traditional procedural languages. Thus, to determine the size that occupies the value of an ASCII string, it is required to look through the entire its contents to find the null terminating character.

The dynamic data types can't be used as the types of variables, at least if this type is mutable, because assignment of a new value to a sub-element of a variable may cause a change in its size, and no compiler can effectively support this kind of operations. The modern string data types, which look like dynamic for the programmer, use dynamic memory to store actual data, and the variables of the types themselves are of fixed size, i.e they are static in terms of our classification. The good example of the complex dynamic data structures, that we can encounter in the modern imperative languages, is the run-time

type information (RTTI). The RTTI is located in the constant sections of the executable files and its representation supports compact encoding of identifiers and some tables of variable lengths.

But when it comes to storing the information in binary files, the dynamic data types become a natural choice. That's why they are so important for the specifications of binary file formats. If we ignore the need to adhere to the requirements to the types of variables of imperative languages, then we can naturally support descriptions of fairly complex dependencies between data elements in the language. We use this approach in the design of the FlexT language.

5.2 Parameters and properties of data types

Data types can have a number of properties, the set of which depends on the kind of the type. For example, the size and the number of elements are the properties of arrays, and the selected case number is the property of variants. Each data type has the property **Size**. The values of the properties can be specified in the statements of type declaration, and also by expressions that compute the value of this property using the values and properties of the nested data elements, and using the values of the parameters of the type. The parameters in the type declaration represent the information that needs to be specified additionally when the type is used (called).

5.3 FlexT data types

The main data types of the language FlexT are shown in the Table 1. The footnote ^a marks the data types, which are supported by the current version of the data reader code generation algorithm.

5.4 Blocks with additional information about data type

The language FlexT has the statements, which are designed to define various kinds of data types (array, record, variant, etc.), of their own syntax. At the same time, there are many tasks that need to be addressed for all the types, regardless of their kinds. For this purpose the language has the blocks of additional information, that can be placed using the symbol ':' after any type definition. The most important of the blocks are:

- the block of assignments intended to specify the values of the data type properties and the parameters of its sub-elements;
- the block **assert**, that we use to specify the condition of correctness of the data type value;
- the block **displ**, that specifies the way, the data type value should be displayed;
- the blocks **let**, that can define new computed properties of the data type.

Table 1: The FlexT data types

Type	Example	Description/purpose
Integer ^a	<code>num-(6)</code>	Differ by the size and the presence of a sign
Empty ^a	<code>void</code>	The type of size 0, marks a place in memory
Characters ^a	<code>char, wchar, wcharr</code>	In the selected character encoding or Unicode with the byte orders LSB or MSB
Enumeration ^a	<code>enum byte (A=1,B,C)</code>	Specifies the names of constants of the basic data type
Term enumeration	<code>enum TBit8 fields(R0: TReg @0.3,...) of(rts(R0) = 000020_ ,...)</code>	Simplifies description of encoding of machine instructions, specifies the bit fields, the presence of which is determined by the remaining bits of the number
Set of bits ^a	<code>set 8 of (OLD ^ 0x02, ...)</code>	Gives the name to bits, the bits can be designated by their numbers (the symbol '=' after the name) or masks (the symbol '^')
Record ^a	<code>struct Byte Len array[@.Len] of Char S ends</code>	Sequential placement in memory of named data elements, which may have different types
Variant ^a	<code>case @.Kind of vkByte: Byte else ulong endc</code>	Selects the content type by the external information
Type check ^a	<code>try FN: TFntNum Op: TDVIOp endt</code>	Selects the content type by internal information (the first type, which satisfies its correctness condition)
Array ^a	<code>array[@.Len] of str array of str ?@[0]= 0!byte;</code>	Consecutive placement of the constituent parts of the same type in memory (the sizes of which may vary). It may be limited by the number of elements, the total size, or the stop condition
Raw data ^a	<code>raw[@.S]</code>	Uninterpreted data, which is displayed as a hex dump
Alignment ^a	<code>align 16 at &@;</code>	Skips unused data to align the next data element at the offset from the base address, which is a multiple of the specified value
Pointer	<code>^TTable near=DWORD, ref=@:Base+@;</code>	Uses the value of the base type for specifying the address (for files – the file offset) of the data of the referenced type in memory
Forward declaration ^a	<code>forward</code>	Makes it possible to describe cyclic dependencies between data types
Machine instructions	<code>codes of TOpPDP ?(@.Op >=TWOpcodes.br) and...;</code>	Machine code disassembling

^aSupported by the data reading code generator

6 Specifications of data formats

Let us consider some examples of data formats described in FlexT, which demonstrate some capabilities of the language.

6.1 The STL files

The STL (from STereo-Lithography) file format [4] is the main mesh representation format for 3D-printing. The possible reason for this is that the STL format is extremely simple. Thus it can be used to quickly and concisely illustrate the FlexT capabilities.

The Listing 1.1 shows the specification of the STL format in FlexT. The STL format has binary and text versions. The binary version has no signature or magic value, that could be used to check whether the file is correct, and to distinguish the binary STL files from the text STL files. Instead, the binary STL files start with an arbitrary 80-char header. Fortunately, the text version should start with the keyword **solid**, so we can split the first 5 chars from the header and check, that they don't contain the text "solid" using the **assert** statement.

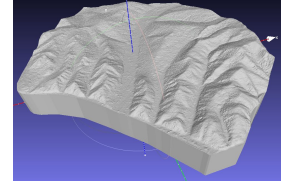


Fig. 1: A 3D model shown from an STL file

Listing 1.1: The STL format specification in FlexT

```
data
0 array[5] of char Hdr0
assert not(Hdr0='solid');

include Float.rfi

type
TSTLPoint array[3] of TSingle
TSTLFace struc
    TSTLPoint Normal
    array[3] of TSTLPoint Vertex
    Word Attr
ends

data
5 array[75] of char Hdr1
80 ulong Count

assert 84+Count*TSTLFace:Size=
    FileSize;

data
84 array[Count] of TSTLFace Faces
```

```
0000000:Hdr0: #Tf1_12_c3 = 'STL f'
0000005:Hdr1: #Tf1_119_c3 = 'file generated by TIN
Smith
0000050:Count: ulong = 000D445C
0000054:Faces: #Tf1_126_c4 = (
0: (Normal: (0:0.125397370586395,1:-0.228255271911621,
2:0.965493440628052);
Vertex: (
0: (0:-14.1680641174316,1:47.9845542907715,
2:10.9498138427734),
1: (0:-14.1677465438843,1:48.1474494934082,
2:10.9882831573486),
2: (0:-14.3192071914673,1:47.9848518371582,
2:10.9695129394531); Attr:0000),
1: (Normal: (0:0,1:0,2:-1);
Vertex: (0: (0:-14.1680641174316,1:47.9845542907715,2:0),1:
(0:-14.3192071914673,1:47.9848518371582,2:0),
2: (0:-14.1677465438843,1:48.1474494934082,2:0)); Attr:0000),
2: (Normal: (0:-0.0630344226956367,1:-0.224894672632217,
2:0.972342014312744);
Vertex: (
0: (0:-14.0169200897217,1:47.984260559082,
2:10.9595441818237),
1: (0:-14.0166034698486,1:48.1471557617188,
2:10.9972410202026),
2: (0:-14.1680641174316,1:47.9845542907715,
2:10.9498138427734); Attr:0000),
3: (Normal: (0:0,1:0,2:-1);
Vertex: (0: (0:-14.0169200897217,1:47.984260559082,2:0),
1: (0:-14.1680641174316,1:47.9845542907715,2:0),
2: (0:-14.0166034698486,1:48.1471557617188,2:0)); Attr:0000),
4: (Normal: (0:0.0398440323770046,1:-0.306879460811615,
```

Fig. 2: An STL file parse results

The FlexT library has no predefined floating point data types. So, the single precision (4-byte IEEE) floating point data type **TSingle** specification is

taken from the file `Float.rfi`. The 3-D vectors are represented by the array `TSTLPoint`. The mesh face (always triangle) is represented by the `TSTLFace` record, that contains the face normal, three vertices and two bytes of attributes.

The file header is followed by the total face `Count` and the array of `Faces`.

The second `assert` checks, that the file size corresponds to the face count.

The Fig. 2 shows the initial fragment of the parse results for the STL file from the Fig. 1 using this specification. Here we can easily understand the file data, so using the FlexT specification the binary file becomes absolutely transparent.

6.2 The HiSCORE custom file format specification

The HiSCORE custom file format is designed to represent the readings of the TAIGA-HiSCORE instrument [5]. The information about the file format was taken from the program `read_hisc.c`, which prints the contents of the HiSCORE files. The resulting FlexT specification is shown in the Listing 1.2.

Listing 1.2: The complete HiSCORE format specification in FlexT

```
type bit
TBit5 num+(5):displ=(int(@))
TBit6 num+(6):displ=(int(@))
TDNS num+(7):displ=(int(@*10))
TBit10 num+(10):displ=(int(@))
TTime struc
    TDNS dns
    TBit10 mks
    TBit10 mls
    TBit6 s
    TBit6 m
    TBit5 h
    num+(20) rest
ends:displ=(int(@.h),':',int(@.m),'',int(@.s),',',int(@.mls),',',int(@.mks),',',int(@.dns))

type
TVal num+(2):displ=(int(@))
TTrackInfo struc
    word offset //track offset
    TVal N //length N
    array[0..N]of TVal Data //N bytes - track data
ends:displ=('[' ,ADDR(&@),']',@)

TPkgData(Sz) struc
    array[9]of TTrackInfo Tracks
    ulong Stop //4 bytes - FF FF FF FF - package end
    raw[] rest //Just in case
ends:[@:Size=@:Sz]:assert[@.Stop=0xFFFFFFFF]

TPkgHdr struc //Package header (24 bytes):
    word idf //data type id = 3008
    word NumBytes //package size (without the 24 bytes of the header)
    ulong NumEvent //event counter number
    ulong StopTrigger //position of stop trigger in DRS counts
    TTime EventT //event time
    word IP //IP address
    word NumSt//Number of station
    TPkgData(@.NumBytes) Data
ends:assert[@.idf=3008]

data
0 array of TPkgHdr:[@:Size=FileSize] Hdr
```

The section **type bit** contains the declarations of the bit-oriented types, which are used to describe the time representation in the HiSCORE files by the **TTime** bit record.

From the top-level view the HiSCORE file is a sequence of packages, represented by the **TPkgHdr** record. So, the size of the array of packages is set to the **FileSize** value. Each package contains a header of constant size and a data block of the size specified in the **NumBytes** header field. The main content of the package data block is the 9 tracks (the **TTrackInfo** data type), each track contains the value count **N** and a sequence of **N** values.

So, the field **NumBytes** is in fact redundant, because this size could be computed from the values of the **TTrackInfo.N** fields. To check, that these values match the **TPkgData** record has an extra field **rest**: this field should normally be of zero size, and the zero-size record fields are not shown by default. But if something goes wrong, we will see a hex dump here.

```
000000:Hdr: #If2_147_c3 = (
0:(idf:0BC0; NumBytes:2128; NumEvent:00002F32;
  StopTrigger:0000014B; EventT:13:19'5.436 166 19; IP:0000;
  NumSt:0000;
  Data:(Tracks: (
    0:[18](offset:00A0; N:400;
      Data: (0:6693,1:6976,2:6914,3:6960,4:6717,5:6805,6:7074,
        7:7178,8:7071,9:7278,10:6986,11:7237,12:6934,13:7093,
        14:6821,15:6945,16:6810,17:7041,18:6909,19:6998,20:7079,
        21:7054,22:7003,23:6983,24:7067,25:7050,26:6951,27:7120,
        28:6899,29:6878,30:6891,31:7069,32:6909,33:7040,34:6900,
        35:6879,36:6817,37:6897,38:6926,39:6965,40:7022,41:6878,
        42:7059,43:6885,44:7122,45:6935,46:7022,47:6920,48:7142
```

Fig. 3: The results of parsing a HiSCORE file

The Fig. 3 shows an initial fragment of the results of parsing a HiSCORE file.

read_hisc.c - 278 lines, Hiscore.rfi - 47 lines

6.3 Weather data in the MM5 format

One of the possible sources of information about a file format is the source code, which can process (read or write) files of this format. The advantages of the source code over the descriptions in natural language are its proved correctness (the code can indeed process the data) and the lack of ambiguity. So, it may seem that understanding a file format by analyzing a source code for its processing will always be easy and preferable to reading the specifications in natural language. The only expected problem is that it may be hard to filter out the data reading/writing code from the much bigger source intended for some particular data processing purposes.

In our experience of FlexT usage we have an indicative example, which demonstrates, that sometimes it may be very hard to understand the file format using the source code.

The file format MM5 is used for representation of the weather forecast data, computed by the same-name Earth climate model [6]. The data contain multidimensional grids for the various climate values (temperature, pressure, wind speed and so on). It was required to read the MM5 file to do something useful with it (say, compute the isolines). To understand the file format we had the source code of a Fortran program for reading and presenting the MM5 data. The Listing 1.3 contains some excerpts from this program. While this code looks very common and does not contain any unexpected statements, no C programmer succeeded in translation it into the C language, because the resulting code, which definitely contained all the analogs of the corresponding Fortran operators, could not read the data correctly.

Listing 1.3: Excerpts from the file readv3.f for reading the MM5 data version 3

```

program readv3
! This utility program is written in free-format Fortran 90.
...
integer, dimension(50,20) :: bhi
real, dimension(20,20) :: bhr
character(len=80), dimension(50,20) :: bhic
character(len=80), dimension(20,20) :: bhrc
character(len=120) :: flnm
integer :: iunit = 10
...
print*, 'flnm=□', trim(flnm)
open(iunit, file=flnm, form='unformatted', status='old',
      action='read')
...
read(iunit, iostat=ierr) flag
do while (ierr == 0)
  if (flag == 0) then
    read(iunit,iostat=ierr) bhi, bhr, bhic, bhrc
...
    call printout_big_header(bhi, bhr, bhic, bhrc)
  elseif (flag == 1) then
...

```

After writing the FlexT specification of the MM5 v3 file format and parsing some files it became obvious that the files contain some additional bytes, which prevent it from reading in C. The additional 4-byte values match the sizes of the data written or read by each of the read/write Fortran operators.

The further investigation revealed, that Fortran **write** operators for binary files consider the list of their arguments as a block, and add information about the size of the block to the file. The four bytes of the size are written before each such block and also after it. The corresponding Fortran read operators expect to

find the block sizes in the file and correctly skip them when reading. They can also check the correctness of the file by comparing the argument list sizes to the block sizes from the file.

As a result, even the Fortran programmers may be unaware about the Fortran binary data blocks representation details. Of course, they should know, that it is required to read the data grouped the same way, as they were grouped when written, but no more. For example, consider the following tutorial about the Fortran file IO [7]. Will You learn the details about the data blocks and their representation in the file memory from this page? And the programmers, who don't use Fortran, are completely clueless about this strange feature.

Note that the case in question dealt with the Fortran language, which is very ancient, but still alive. And now imagine what may happen if some cryptic esoteric language would be used instead.

The Listing 1.4 contains the fragment of the MM5 file format specification in FlexT, which corresponds to the last read operator in the Listing 1.3.

Listing 1.4: Excerpt from the MM5 data version 3 format specification in FlexT

```
TBHi array[50] of array[20] of i4
Tbhr array[20] of array[20] of TReal

TComment array[80] of Char, <0x20;

TBHiC array[50] of array[20] of TComment
TbhrC array[20] of array[20] of TComment

TBigHeader struct
    u4 BHSIZE //Size of Data - added automatically by Fortran write
    TBHi BHi
    Tbhr bhr
    TBHiC BHiC
    TbhrC bhrC
    u4 BHSIZE_ //Size of Data - added automatically by Fortran write
ends: assert[@.BHSIZE==@:size-8, @.BHSIZE_=@:size-8]
```

The record **TBigHeader** corresponds to the single Fortran read/write block. It explicitly contains the fields **BHSIZE** and **BHSIZE_** with the values of the block size added by the Fortran IO operators. This specification discloses the peculiarities of the Fortran system library and makes them evident to the programmers in any programming language. And it is easy to understand this human-readable format specification even without any prior knowledge about the language FlexT. So, the text of specification itself can help the future digital archaeologists to parse the MM5 files.

7 Code generation

7.1 Generation of the data reading code

The format specifications are required to write a correct program, that should work with the files of the format. Because the FlexT language data types look

similar to that of imperative languages, it is possible to immediately use some parts of specification to declare the data types, constants, and so on, which are required to write the data processing code. Anyway the process of writing the code manually is still time-consuming and error-prone. So, we have implemented the code generator, which can automatically produce the data reading code in imperative languages from the FlexT specifications. By its expressive power the FlexT language outperforms the other projects developing the binary format specifications, so the task of code generation for the FlexT specifications is rather nontrivial, because it is required to reconstruct all the features in the generated code. By now we have implemented the code generation for the most widely used FlexT data types, but some complex types are not supported yet. The main principles and algorithms we use for code generation from FlexT specifications were considered in [8].

The Listing 1.5 contains an excerpt from the ESRI Shape file format specification – a data type **TArcData** definition. The Listing 1.6 shows the code of the method for getting the field **T** from the accessor (i.e. the auxiliary class generated for reading the complex data type) of the data type of the field **TArcData.Points**. The code uses the values from the array **Parts** to determine the number of elements in the array **T**, and the **if** operator was generated for the expression of the **TXPointTbl** parameter.

Listing 1.5: FlexT specification of polygon/polyline data in Shape file format

```
TArcData struct
  TBBBox BBox
  long NumParts
  long NumPoints
  array[@.NumParts] of long Parts
  array[@.NumParts] of struct
    TXPointTbl((@@@.Parts[@:#+1] exc @@@.NumPoints)-@@@.Parts[@:#]) T
  ends Points
ends
```

7.2 Generation of the test application

The first thing any programmer will want to do after generating a data reader is to test whether it works well. To perform the test it is required to write an application, which will use the data reader somehow. The most obvious and illustrative task here is to print using the data reader. Having manually created several test programs of this kind, we found that this process is rather tedious and should be automated. So, we have developed the algorithm, which automatically generates the test code. The test program generated together with the data reader makes it possible to immediately check the reader. Of no less importance is the fact that the source code of the program demonstrates the main patterns of data access using the reader. The Listing 1.7 shows some fragments of the test application code automatically generated for the ESRI Shape file format.

Listing 1.6: Generated Pascal code, which provides accessor for the field T

```
function TTArcData_Sub1Accessor.T: TTXPointTblAccessor;
var
  i0: Integer;
  ndx0: Integer;
begin
  if not Assigned(FT) then begin
    ndx0 := Index+1;
    if (ndx0>=0) and (ndx0<TTArcDataAccessor(TTArcData_Sub2Accessor(Parent).
      Parent).Parts.Count) then
      i0 := TTArcDataAccessor(TTArcData_Sub2Accessor(Parent).Parent).Parts.
        Fetch(ndx0)
    else
      i0 := TTArcDataAccessor(TTArcData_Sub2Accessor(Parent).Parent).
        NumPoints;
    FT := TTXPointTblAccessor.Create(Self,0,0,i0-
      TTArcDataAccessor(TTArcData_Sub2Accessor(Parent).Parent).Parts.Fetch(
        Index));
  end;
  Result := FT;
end ;
```

Listing 1.7: Fragments of the test application code in C++, immediate write style

```
std::unique_ptr<TSHPReader> must_free_Reader(new TSHPReader(FN));
Reader = must_free_Reader.get();
if (!AssertTShpHeader(Reader->Hdr(),Reader))
  exit(2);
cout<<"Hdr:"<<endl;
cout<<sIndent<<"Magic:_"<<Reader->Hdr()->Magic.Value()<<endl;
...
cout<<sIndent<<"FileLength:_"<<Reader->Hdr()->FileLength.Value()<<endl;
cout<<sIndent<<"Ver:_"<<Reader->Hdr()->Ver<<endl;
...
cout<<"Tbl:"<<endl;
for (i=0; i<Reader->Tbl()->Count(); i++) {
  V = Reader->Tbl()->Fetch(i);
  cout<<sIndent<<"["<<i<<"]:"<<endl;
  cout<<sIndent<<"RecNo:_"<<V->RecNo()<<endl;
  cout<<sIndent<<"Len:_"<<V->Len()<<endl;
  if (!V->Data()->GetAssert())
    exit(2);
  cout<<sIndent<<"Data:"<<endl;
  cout<<sIndent<<"ST:_"<<TShapeTypeToStr(V->Data()->ST())<<endl;
  cout<<sIndent<<"SD:"<<endl;
  switch ( (TShapeRecData_Sub0_Case)V->Data()->SD()->hCase() ) {
    case hcPoint:
      cout<<sIndent<<"Point:"<<endl;
      cout<<sIndent<<"X:_"<<V->Data()->SD()->cPoint()->X<<endl;
      cout<<sIndent<<"Y:_"<<V->Data()->SD()->cPoint()->Y<<endl;
      break;
    ...
    case hcMultiPointZ:
      cout<<sIndent<<"MultiPointZ:"<<endl;
      ...
      cout<<sIndent<<"Points:"<<endl;
      for (i13=0; i13<V->Data()->SD()->cMultiPointZ()->A()->Points()->Count()
        ; i13++) {
        V13 = V->Data()->SD()->cMultiPointZ()->A()->Points()->Fetch(i13);
        cout<<sIndent<<"["<<i13<<"]:"<<endl;
        cout<<sIndent<<"X:_"<<V13->X<<endl;
        cout<<sIndent<<"Y:_"<<V13->Y<<endl;
      }
  }
}
```

8 Conclusion

We have considered the possible options, which should be examined when selecting a file format for scientific data representation.

The formal specifications of binary file formats, especially for the custom ones, are very important, because the natural language specifications are ambiguous, and it may be hard to fetch the data format information from the source code (the MM5 file format case demonstrates it very well).

The language FlexT is designed to write compact, human-readable and powerful specifications, which can be used to check the correctness of data and resolve the ambiguities in the understanding of the other kinds of information about file formats.

It is also possible to generate from the FlexT specification the data reading code and the code of the application, that can immediately test the generated reader by printing the whole content of a binary file according to the specification using the reader. The current level of capabilities of the code generator is well characterized by the fact that it has successfully produced a full-featured code for reading data for the well-known in the GIS community Shape file format. The FlexT specification of the Shape format takes approximately 180 lines of code. The code generator have produced 1570 lines of the reader code, and 375 lines of the test program. The algorithm described in the present paper has also been used for the generation of the data readers for the scientific file formats of the experiments in the TAIGA observatory.

References

1. The HDF Group. Income accessed online on 15th April 2019 via <https://www.hdfgroup.org>
2. Protocol Buffers. Income accessed online on 15th April 2019 via <https://developers.google.com/protocol-buffers/>
3. Hmelnov A., Bychkov I., Mikhailov A. A declarative language FlexT for analysis and documenting of binary data formats. Proceedings of the Institute for System Programming, vol. 28, issue 5, 2016, pp. 239-268. (in Russian) DOI:[http://dx.doi.org/10.15514/ISPRAS-2016-28\(5\)-15](http://dx.doi.org/10.15514/ISPRAS-2016-28(5)-15)
4. The StL Format. Income accessed online on 15th April 2019 via <https://www.fabbers.com/tech/STL.Format>
5. Bychkov, I.; Demichev, A.; Dubenskaya, J.; Fedorov, O.; Haungs, A.; Heiss, A.; Kang, D.; Kazarina, Y.; Korosteleva, E.; Kostunin, D.; Kryukov, A.; Mikhailov, A.; Nguyen, M.-D.; Polyakov, S.; Postnikov, E.; Shigarov, A.; Shipilov, D.; Streit, A.; Tokareva, V.; Wochele, D.; Wochele, J.; Zhurov, D. Russian–German Astroparticle Data Life Cycle Initiative. Data 2018, 3, 56.
6. MM5 Community Model. Income accessed online on 15th April 2019 via <http://www2.mmm.ucar.edu/mm5/>
7. Fortran – File Input Output. Income accessed online on 15th April 2019 via https://www.tutorialspoint.com/fortran/fortran_file_input_output.htm
8. Hmelnov, A., Mikhailov, A. Generation of code for reading data from the declarative file format specifications written in language FlexT // IEEE The Proceedings of the 2018 Ivannikov ISPRAS Open Conference (ISPRAS-2018) 2018. P. 9-15.